# Efficient Lifted Planning with Regression-Based Heuristics

by

Hadi Qovaizi

Bachelor of Science, Ryerson, 2016

A thesis

presented to Ryerson University

in partial fullfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Efficient Lifted Planning with

Regression-Based Heuristics

Master of Science 2019

Hadi Qovaizi

Computer Science

Ryerson University

## Abstract

Modern state-of-the-art planners operate by generating a grounded transition system prior to performing search for a solution to a given planning task. Some tasks involve a significant number of objects or entail managing predicates and action schemas with a significant number of arguments. Hence, this instantiation procedure can exhaust all available memory and therefore prevent a planner from performing search to find a solution. This thesis explores this limitation by presenting a benchmark set of problems based on Organic Chemistry Synthesis that was submitted to the latest International Planning Competition (IPC-2018). This benchmark was constructed to gauge the performance of the competing planners given that instantiation is an issue. Furthermore, a novel algorithm, the *Regression-Based Heuristic Planner* (RBHP), is developed with the aim of averting this issue. RBHP was inspired by the retro-synthetic approach commonly used to solve organic synthesis problems efficiently. RBHP solves planning tasks by applying domain-independent heuristics, computed by regression, and performing best-first search. In contrast to most modern planners, RBHP computes heuristics backwards by applying the goal-directed regression operator. However, the best-first search proceeds forward similar to other planners. The proposed planner is evaluated on a set of planning tasks included in previous International Planning Competitions (IPC) against a subset of the top scoring state-of-the-art planners submitted to the IPC-2018.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation and Research Problem

Classical planning is the subfield of planning that aims to solve a specific subset of problems. A given planning task specifies the initial set of facts, the desired set of goal facts, and the actions that manipulate objects within the environment. Classical planning is performed using deterministic actions in given environments which are characterized explicitly. The objective is to determine a sequence of actions which can be applied from the initial state in order to satisfy the goal. Identifying a solution is mainly challenging due to the combinatorics of applying actions which results in an exponentially large state-space in the worst possible case [69]. In fact, the complexity of planning is PSPACE-complete [6]. Additional challenges may emerge from the given planning task as well. Some tasks require intricate and lengthy solutions to satisfy the goal. Furthermore, modern planners generally conduct planning by instantiating the grounded transition system in advance, so that path-finding algorithms can be applied to find a solution. However, this grounding procedure may exhaust all available memory resources and in turn prevent the planner from performing search.

Furthermore, the initial state of a given planning task is considerably large in general. Meaning that, the initial state normally contains a large number of objects, some of which may be irrelevant in solving the task. Moreover, the problem domain may entail a significant number of actions. In turn, more actions may be applicable in forward search and therefore increase the complexity of determining a (optimal) solution to the problem. In contrast, the goal state is generally dense and consists of closely related facts. Thus, the set of actions that must have been applied to directly

produce the goal are much smaller in comparison.

Naturally, searching backwards from the goal towards the initial state offers potential advantages over forward search. Provided that backward search can moderate the search effort, we consider regression-based strategies in planning. These approaches apply divide-and-conquer techniques by decomposing the goal into sub problems that are more manageable to solve. These algorithms operate effectively by ignoring the interaction between goals. As a result, regression-based approaches can compute informative heuristics inexpensively to better direct search. Nevertheless, regression-based methods share a number of drawbacks, such as: the production and management of incomplete plans, searching from partial states, and management of invalid states. In this thesis, we explore regression-based approaches in planning and demonstrate our work by developing a novel domain-independent heuristic planner, namely the *Regression-Based Heuristic Planner* (RBHP).

## 1.2 Methodology

To further emphasize the issues associated with grounding, we develop the Organic Chemistry Synthesis benchmark. The set of problems is based on 20 organic synthesis examination questions from the Massachusetts Institute of Technology (MIT). An initial version of this benchmark was developed as a planning problem by Masoumi and Soutchanski [45, 46]. However, a number of reactions were chemically incorrect or incompletely encoded. Henceforth, we have corrected and verified the chemical reactions by consulting with experts in organic synthesis and with established online resources.

In order to conduct planning on this benchmark efficiently, we developed the *BinTree* solver. This solver applies uninformed depth-first iteratively deepening search (DFIDS). BinTree makes use of the binary tree data structure to maintain the state efficiently. Additionally, the solver enables the use of domain-independent heuristics given domains whereby the effects of actions are explicitly characterized without a situation argument.

Furthermore, to increase the efficiency of assessing action preconditions in chemistry, we encoded preconditions as graphs of chemical atoms. Then, we developed an efficient implementation of sub-graph isomorphism to assess the preconditions against the current state. Subsequently, we translated the organic synthesis benchmark from a specialized chem-informatics format to suit the BinTree solver. In turn, we compare the BinTree solver with an uninformed DFIDS planner. This planner maintains states as an implicit set of fluents that include a situation argument whereby the assessment of these fluents is performed recursively. The comparison between solvers was performed to gauge the performance of the new state representation used by the BinTree solver. Additionally, we encoded the problems into the *PDDL* format and submitted the benchmark to the 9th International Planning Competition of 2018. This benchmark was developed to gauge the performance of competing planners given that grounding is a prevalent issue.

In addition, we developed a novel regression-based planner, RBHP, that applies domain-independent heuristics in forward best-first search. This planner computes a goal-directed backward heuristic based on goal-regression. Specifically, actions required to satisfy the goals are identified by focussing on objects mentioned in the goal formula. The actions are then used to populate a look-up table in order to provide heuristics during forward A* search. This planner exploits the advantages of several algorithms such as *Unpop*, *GRT*, *GRAPHPLAN*, and *A* search*, while avoiding their major limitations.

To gauge the performance of RBHP, we encode PDDL domains (used in previous planning competitions) into Prolog by developing a Java *PDDLToProlog* translator. The domains are encoded to correspond with the binary tree representation of states. Finally, we evaluate RBHP on these planning tasks and compare its performance against competing planners submitted to the IPC-2018.

## 1.3 Novelty and Significance

Our interdisciplinary work involving organic chemistry synthesis proves to be beneficial for both fields. The organic synthesis domain was researched given its practicality and real-world application value. Also, applying computer-assisted methods to this field help automate and in turn significantly speed up the procedure. Previously, there was lack of computational tools available

for chemists to perform organic synthesis [58]. Although recently, Szymkuc et al. [67] have successfully made significant progress in joining the two fields. As a result, new chemical pathways may be identified which minimize the cost or difficulty of the process. However the approaches taken within [67] are specific with regards to the chemistry domain. Whereas, AI planning aims to develop domain-independent approaches in order to solve problems in general. With regards to the advantages for planning, this domain emphasises the drawback of the grounding procedure. Also, the complexity and challenges posed by this domain further highlight the limitations exhibited by modern planning. In turn, this domain provides a greater incentive for modern planning to consider and research novel approaches, such as planning in the backward direction.

Moreover, search and the construction of heuristics is predominantly performed in the forward direction from the initial state towards the goal. This trend is primarily due to the lack of thorough research in mediating the limitations exhibited by alternative approaches such as goal-regression [70]. In this thesis, we explore alternative methods of planning related to regression in order to moderate issues demonstrated by modern planners. Our main approach incorporates the advantages of several algorithms while avoiding their main limitations. Furthermore, we address the leading drawbacks to regression-based approaches and make attempts to alleviate them.

## 1.4 Contribution and Outline

With regards to our contribution, we developed and submitted the Organic Chemistry Synthesis benchmark to the 9th IPC-2018 to emphasize issues related to grounding. Whereby, we were awarded the *Outstanding Domain Submission Award* by the organizers of the competition for our effort.

A novel planner, *BinTree*, was developed to efficiently maintain the state of the environment throughout the planning procedure. This solver facilitates the use of domain-independent heuristics by explicitly updating the current state with the effects of the last action applied. We also produced an efficient sub-graph isomorphism implementation responsible for the identification of molecular structures within graphs. As a result, we show that BinTree in junction with sub-graph isomorphism demonstrates considerable improvement in performance by optimally solving 7 out of the 20 organic synthesis problems from our challenging benchmark. These problems

were solved in an average of 60 seconds using a standard desktop computer. Whereas, comparable performance was previously achieved by another research group, namely Heifets and Jurisica [22, 21], given up to 6 hours to solve the problems by using an IBM super-computer.

Furthermore, we investigate approaches that are not commonly used in practice by the planning community. In doing so, we developed the regression-based heuristic planner, *RBHP*. We demonstrate the performance of RBHP against the top three scoring planners from the IPC-2018. These state-of-the-art planners have been developed for over 10 years by teams consisting of senior PhD students and post doctoral researchers. Whereas, RBHP was developed in less than one year. Although the results were compared for a set of problems from one domain, RBHP demonstrates considerable potential.

The remainder of this thesis is organized as follows. Chapter 2 reviews the fundamental concepts and formalisms with regards to classical planning. Subsequently, we summarize well-known path-finding algorithms and advanced strategies that concentrate the search in order to moderate the search effort. We then briefly review the history and development of state-of-the-art planning. We then review in more detail the related work which lead to the production of our main contribution in this thesis.

Chapter 3 discusses the background of Organic Chemistry Synthesis and the methodology of our applications when working with this domain as a planning problem. The novel algorithm, *BinTree*, which represents and maintains states using binary trees is presented. Furthermore, we provide an overview of our efficient sub-graph isomorphism implementation. We present the organic synthesis benchmark based on organic chemistry examination questions from the MIT. The experimental results produced by BinTree are analysed against a comparable algorithm.

Chapter 4 details the implementation of RBHP. A description of the domains used in our experiments is provided. The performance of RBHP is evaluated against state-of-the-art planners whereby the experimental results are presented. We then analyse the performance of RBHP and recognize the limitations exhibited by the algorithm. Chapter 5 concludes the thesis with a discussion about results, limitations, and future work.

## Chapter 2

# Background

In this chapter, we review the fundamentals required to conduct classical planning in Artificial Intelligence (AI), whereby the standard concepts, formalism, and notations are explained. We begin by reviewing the situation calculus (SC) formalism. We discuss classical planning in SC. Afterwards, a concise overview of general and advanced search strategies is presented. Subsequently, we review the development of state-of-the-art planning. Lastly, we present the research and works that motivates the main methodology produced in this thesis.

## 2.1 Planning

In the field of AI, planning is concerned with applying strategies that allow intelligent agents to identify a course of action which satisfies the desired objective. Different classes of planning problems pose a wide variety of challenges. These challenges largely depend on the properties of the given problems which planners aim to solve. Certain classes involve planning on domains which only provide partial information of the environment. Other classes may include actions that have non-deterministic effects, or actions that have a duration. For our purposes, this work focusses on classical planning which aims to solve a specific subset of planning problems.

As in general planning, the task in classical planning is to determine a sequence of actions (i.e., plan), that can satisfy the goal state, from the given initial state. This field applies search strategies using actions with deterministic effects within completely known environments. This field poses two main challenges with regards to scalability. The first challenge is evident given that the state-space may become significantly large, especially for problems that involve a significant number of objects. In turn, more actions may become applicable which increases the complexity

of identifying the correct plan in order to satisfy the goal. The second difficulty is apparent given that exceedingly difficult problems require progressively longer and more intricate sequences of actions in order to satisfy the goal. As a result, more effort is required during search in order to identify the plan as the state-space may become exponentially large.

The following section outlines the fundamental formalization and definitions which lay the groundwork required to perform classical planning.

### 2.1.1 The Situation Calculus (SC)

In order to conduct classical planning, a formal method is required to model the environment and account for the changes that occur in the system. These systems are often dynamic and vary significantly in terms of their complexity. For example, certain domains may be difficult to express given that they involve stochastic actions or concurrent actions with interacting effects. Other domains may involve a hybrid dynamics of continuous and discrete processes. Consequently, the task of formalizing a general method to express these environments is non trivial. The *situation calculus* (SC) [48], introduced by John McCarthy in 1963, provides the foundation for representing and reasoning about discrete dynamic domains in planning. Further development in axiomatizing the SC provides a logic formalism that makes classical planning possible.

SC is an extension of first-order logic (FOL) [38]. FOL specifies the set of all well-formed formulas that can be composed using a selected set of symbols called the vocabulary of the language. The vocabulary consists of the set of logical symbols and parameters. The logical symbols include parenthesis, logical connectives, variables, and equality. The parameters include the quantifier symbols, predicate symbols and function symbols. Specifically, the logical symbols $\wedge$ and $\vee$ are used to represent the connectives for conjunction and disjunction, respectively. Furthermore, the symbol $\rightarrow$ and $\leftrightarrow$ represent implication and equivalence, correspondingly. The symbols $\forall$ and $\exists$ signify the universal and existential quantifiers, respectively. Additionally, $\cup$ represents the union of sets and $\neg$ stands for negation. The symbols beginning with capital letters denote constants. The symbols beginning with lower case letters represent variables, with the exception of function symbols which are constants. All free variables are universally quantified $\forall$ at the front implicitly in axioms.

The SC language consists of three disjoint sorts: actions, situations, and objects. Actions are used to transition from a given situation $s$ to a resulting situation $s'$. Actions are represented as terms, $A(\overrightarrow{x})$, whereby $\overrightarrow{x}$ is a tuple of variables that are instantiated by distinct objects from the given domain. Actions have preconditions that must be satisfied in order to become applicable. Once all arguments of a given action are instantiated with constants, it is considered to be grounded and applicable in the current situation if its preconditions are true. Furthermore, actions have direct effects that become true as a result, or become false as a consequence of the action being executed. For instance, *stack(BlockA, BlockB)* represents the action of placing a block *BlockA* onto another block *BlockB*. Correspondingly, *unstack(BlockA, BlockB)* represents the action of lifting *BlockA* from *BlockB*.

A *situation*, in FOL, corresponds to the sequence of actions that have been applied, starting from the initial situation $S_0$. The function *do(a, s)* is used to represent the resulting situation from applying action $a$ in situation $s$. For instance, *do(stack(BlockA, BlockB), do(pick-up(BlockA), $S_0$) )* represents the situation where block *BlockA* is picked up from the table, and then stacked on top of block *BlockB*. In this way, the history of the actions is maintained. Formally, s $\prec$ s' represents that a situation $s$ is a proper subhistory of another situation $s'$ if the sequence of actions in $s$ are included in $s'$ and not equal to the sequence in $s'$.

Fluents $F(\overrightarrow{x}, s)$ represent varying properties that are dependant on the given situation $s$. Relational fluents are predicates that denote when a condition is true. For example, the relational fluent *On(BlockA, BlockB, s)* is true if *BlockA* is directly on *BlockB* in situation $s$. Functional fluents are functions which provide a value for a given argument within $\overrightarrow{x}$. For example, the functional fluent *beside(BlockA, s)* returns a value for a block that is next to *BlockA* in situation $s$.

A logical theory that is used to capture the dynamics of a domain in SC is a basic action theory (BAT) [62]. Specifically, the BAT is a set of axioms that models the initial knowledge base as well as actions, the preconditions required to apply them, and the effects that occur as a result of performing them. A basic action theory $D$ is defined to be the union of the following five axioms:

$$D = \Sigma \cup D_{ss} \cup D_{ap} \cup D_{una} \cup D_{s0}$$

- $\Sigma$ are the four foundational axioms for situations as follows:

$$do(a_1, s_1) = do(a_2, s_2) \leftrightarrow a_1 = a_2 \wedge s_1 = s_2 \tag{2.1}$$

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \leftrightarrow P(do(a, s))] \leftrightarrow (\forall s)P(s) \tag{2.2}$$

$$\neg s \prec S_0 \tag{2.3}$$

$$s \prec do(a, s') \leftrightarrow s \preceq s' \tag{2.4}$$

  whereby axiom 2.1 is a unique names axiom for situations. Axiom 2.2 is a second-order induction axiom, that together with axiom 2.1 implies two situations will be the same if and only if they result from the same sequence of actions applied to the initial situation. Axiom 2.3 denotes that there is no situation that is a proper subhistory of the initial situation. Finally, axiom 2.4 denotes that a situation $s$ is a proper history of *do(a, s')* if and only if $s$ is a history of $s'$.

- $D_{ss}$ denotes the set of successor state axioms (SSAs). The SSAs characterize the set of effects which become true as a result, or become false as a consequence of the most recent action applied. Formally, for each fluent $F(\overrightarrow{x}, s)$, there is one SSA axiom of the following form:

  $F(\overrightarrow{x}, do(a, s) \leftrightarrow \Phi_F(\overrightarrow{x}, a, s)$

  where $\Phi_F(\overrightarrow{x}, a, s)$ is a formula uniform in $s$, whereby all free variables are among $a$, $s$, $\overrightarrow{x}$. A situation calculus formula $\psi(s)$ is uniform in $s$, if the formula has no occurrences of the predicate *Poss*, $\prec$, has no quantifiers over the situation, and $s$ is the only situation term mentioned in $\psi(s)$.

  For each fluent, the SSA has the following general form:

  $F(\overrightarrow{x}, do(a, s)) \leftrightarrow \bigvee_i a = PosAction_i(\overrightarrow{x}) \wedge \gamma_i^+(\overrightarrow{x}, s) \vee$

  $F(\overrightarrow{x}, s) \wedge \neg(\bigvee_j a = NegAction_j(\overrightarrow{x}) \wedge \gamma_j^-(\overrightarrow{x}), s))$.

The SSA dictates that a fluent becomes true upon the execution of an action if the action causes it to become true (i.e., $PosAction_i$), or the fluent remains true given that it was true and the action did not cause it to become false. The fluent becomes false if the action applied causes the fluent to become false (i.e., $NegAction_j$). Otherwise, the fluent remains unaffected. Note that $\gamma_i^+(\overrightarrow{x}, s)$ is the context formula that expresses the conditions which must hold in $s$ whereby the positive effect can occur. Similarly, $\gamma_j^-(\overrightarrow{x}, s)$ is the context formula that expresses the conditions that must hold in $s$ whereby the negative effect can occur. Given that the most recent action applied does not make the fluent true or false, then the fluent remains unaffected. For instance, consider the successor state axiom for the fluent *On*, as follows:

$$On(blockA, blockB, do(a, s)) \leftrightarrow a = stack(blockA, blockB) \vee$$

$$(On(blockA, blockB, s) \wedge a \neq unstack(blockA, blockB)).$$

This SSA states that a block *blockA* is directly on top of another block *blockB* if the most recent action applied was to stack *blockA* onto *blockB*, or *blockA* was already on top of *blockB* in situation $s$, and the most recent action aplied was not to unstack *blockA* from *blockB*.

- $D_{ap}$ represents the set of action precondition axioms (PAs). The PAs define the set of requirements that must be satisfied in order to perform an action. Formally, for an action term $A(\overrightarrow{x})$, the binary predicate $Poss(A(\overrightarrow{x}), s)$ is true if the action can be applied in situation $s$. For each action, there is one PA axiom of the form $Poss(A(\overrightarrow{x}), s) \leftrightarrow \Pi_A(\overrightarrow{x}, s)$. Whereby, $\Pi_A(\overrightarrow{x}, s)$ denotes the condition that must hold in situation $s$ in order for the action to become applicable in $s$. For instance, the following precondition axiom states that it is possible to pick up a block *blockA* if and only if *blockA* is clear, *blockA* is on the table, and the hand is empty in the situation $s$:

  $$Poss(pick\text{-}up(blockA), s) \leftrightarrow Clear(blockA, s) \wedge OnTable(blockA, s) \wedge HandEmpty(Arm, s).$$

- $D_{una}$ represents the set of unique names axioms for actions and objects. In other words, two actions with distinct names represent two distinct actions.

- $D_{S_0}$ is the set of axioms that characterizes the initial theory $S_0$ of a given problem, prior to the application of any action. This set consists of FOL formulas that include the initial situation $S_0$ as the only situation argument. In addition to sentences that mention the situation argument $S_0$, an initial database may include sentences that do not include $S_0$. Meaning that, the initial theory may include sentences that are independent of the situation. For example, *Block(BlockA)* represents that *BlockA* is a type of block.

In addition to fluents which are directly affected by the actions executed in a situation, there exists another set of fluents that are affected by the actions implicitly. This set is known as *derived predicates*. These predicates are not defined by SSAs. Instead, they are defined recursively from other fluents using formulas known as *state constraints*. For instance, the predicate *Above(blockA, blockB, s)* is true if *blockA* is directly on top of *blockB* or *blockA* is directly on top of some *blockC* such that *blockC* is above *blockB*:

$$Above(blockA, blockB, s) \stackrel{\text{def}}{=}$$
$$( On(blockA, blockB, s) ) \vee \exists blockC( On(blockA, blockC, s) \wedge Above(blockC, blockB, s) ).$$

The validity of this predicate is dependent on the state constraints as no action can directly make it true or false. However, actions directly affect the fluent *On(blockA, blockB, s)* which in turn has an indirect effect on the predicate *Above(blockA, blockB, s)*. Derived predicates facilitate the modularization of the logical theory. The precondition axioms can make use of derived predicates. The derived predicates may also be used as part of the goals. In turn, the axioms can be defined concisely and more conveniently.

### 2.1.2 Classical Planning

Classical planning is the task of generating a sequence of actions that satisfies the goal given a complete characterization of a domain. In classical planning, there are two main sub-areas: 1) planning for specific domains, and 2) domain-independent planning. The focus of thesis is on domain-independent planning. Although, planning is computationally intractable across both classes of domains, domain-independent planning is additionally challenging since the strategies

used must be applicable in general. Meaning that domain-independent planning must not use specific information respective to the domain to assist in solving the problem. Consequently, information must be computed automatically to direct the agent in guiding the search. The techniques used by planning algorithms to assist in guiding search are known as *heuristics*. Given an accurate heuristic, less computational effort is required during the search to reach the goal. In turn, the time and memory resources required to perform the search are minimized. An informative heuristic is valuable when solving exceedingly challenging problems.

The state-space is represented by a weighted directed graph, $G = (V, E, w)$, whereby $V$ is the set of vertices, and $E$ is the set of directed edges which are attributed weights $w$, respectively. This graph is used in accordance with the classical model $<S, S_0, SG, A, f, c>$ whereby:

- $S$: is the set of states,
- $S_0$: is the initial state,
- $SG$: is the goal state,
- $A$: is the set of actions,
- $f(a, s)$: is the deterministic transition function, and
- $c(a, s)$: is the positive cost associated with performing an action.

The graph is used to identify a solution for a given problem, if a solution exists. Whereby, the vertices represent states $s \in S$ and the edges $(s, s')$ represent the actions, such that state $s'$ follows state $s$ upon performing the action $a \in A(s)$. Path-finding algorithms can be divided into two groups: 1) uninformed search algorithms, and 2) informed search algorithms. Uninformed search algorithms perform search blindly without making use of heuristics. Examples of uninformed search include *depth-first*, *breadth-first*, and *uniform cost search*. Informed search algorithms are goal-directed by making use of heuristic functions whereby the goal state influences the search. Some examples of these algorithms include *best-first* and *greedy best-first search*.

In general, both classes of search algorithms function similarly by starting at the initial state and advancing the search frontier. The search algorithms typically maintain two lists of nodes, the CLOSED list and the OPEN list. The CLOSED list consist of the nodes that have been previously

expanded by generating their children (i.e., neighbouring nodes). The OPEN list, also known as the *search frontier*, consists of the nodes that have been generated, but not yet expanded. The frontier is extended by selecting a generated node and expanding it. If the selected node does not satisfy the goal, then the node's neighbours are added to the frontier according to the algorithm. The planning procedure continues this cycle until either a solution is found, a prescribed bound (or time limit) is reached, or memory resources are exhausted.

These algorithms generally differ in the method used to represent the search frontier and the method used to select nodes from the frontier. In addition, planning algorithms may vary in the direction of search. Specifically, a search algorithm may perform forward search from the initial state to the goal state. Search may also be performed backwards from the goal state to the initial state. Otherwise, a combination of both forward and backward search may be used to perform bi-directional search.

An algorithm is *complete*, meaning that, it will identify a solution from the initial state to the goal state within a finite amount of time, given such a path exists. Additionally, if a planner determines a plan that is a valid solution to the problem, then it is *sound*. Furthermore, not all algorithms will produce the optimal solution path (i.e., plan with least cost). Optimality is dependent on the heuristic function used by the algorithm. The heuristic function is used to guide the search by computing an estimate of the remaining distance towards the goal from a given state. The requirement for optimality is that the heuristic estimate must be admissible, meaning that it does not overestimate the true optimal cost required to satisfy the goal from a given state. Optimal algorithms apply admissible heuristics in order to produce optimal plans, whereas satisficing algorithms produce plans that are not guaranteed to be optimal.

### 2.1.3 General Search Algorithms

The following serves as a concise review of well known AI path-finding algorithms. These algorithms apply search over directed graphs in an attempt to produce a solution for the classical planning model [18, 4, 35, 34]:

1. *Depth-First Search (DFS)*:

   *DFS* is a method of uninformed search, whereby the search frontier is implemented as a stack. Upon the expansion of the frontier, the neighbouring nodes are pushed to the top of

the stack. *DFS* operates in linear space *O(bd)* where $b$ is the branching factor and $d$ is the length of the optimal solution. *DFS* conserves memory resources since it must only maintain the current node and the ancestors' child nodes which have not yet been expanded. Also, *DFS* can easily prune paths that contain cycles. Cycles are paths in the graph which contain duplicate nodes. *DFS* is not guaranteed to produce the optimal plan. Furthermore, *DFS* may exceed the imposed time limitation given that it exhaustively assesses all nodes in the current branch before proceeding to a different branch. *DFS* performs poorly for problems whereby the goal is shallow respective to the initial node within the graph (i.e., when an optimal plan is short).

2. *Breadth-First Search (BFS)*:

   *BFS* is a type of uninformed search, where by the search frontier is implemented as a queue. Upon the expansion of the frontier, the neighbouring nodes are pushed to the back of the queue. Hence, *BFS* expands all nodes at a given depth prior to expanding nodes at a successive depth. *BFS* operates in exponential space and time $O(b^d)$ where $b$ is the branching factor and $d$ is the length of the optimal solution. *BFS* is more likely to exhaust memory resources than *DFS* as the number of nodes generated and maintained by the search are significantly large. *BFS* will produce the optimal plan if the given action costs are uniform. *BFS* performs well for problems whereby the goal is shallow respective to the initial node within the graph.

3. *Uniform-Cost Search (UCS / Dijkstra's algorithm)*:

   *UCS* is a form of uninformed search that uses the evaluation function $f(n) = g(n)$, where $g(n)$ is the accumulated cost from the initial node to the current node $n$. In other words, *UCS* aims to greedily identify the shortest path between the initial node and goal node. *UCS* maintains the frontier as a priority queue whereby the nodes with minimum cumulative cost from the initial state are given highest priority. In turn, *UCS* will produce the optimal plan, if one exists. However, *UCS* suffers from the same memory limitations as BFS.

4. *Bounded-Cost Depth-First Search (BC-DFS)*:

   *BC-DFS* is an extension of *DFS* whereby the search frontier is implemented as a stack. The evaluation function used by *BC-DFS* is $f(n) = g(n)$, where $g(n)$ is the accumulated cost of reaching the current node $n$ from the initial node. Upon expansion of the frontier during the search, if the cost of reaching a node $g(n)$ exceeds the prescribed bound $B$, then the search is

terminated for that path. In addition, *BC-DFS* identifies and prunes paths that contain cycles. As in *DFS*, *BC-DFS* uses linear space (i.e., *O(bd)*). *BC-DFS* is optimal given that the bound is not less less than the cost $C^*$ of an optimal solution. *BC-DFS* only produces the optimal solution given that $B = C^*$.

5. *Depth-First Iteratively Deepening Search (DFIDS))*:

   *DFIDS* is similar to *BC-DFS* in that a prescribed bound $B$ is imposed. However, the search iterates on the depth by successively increasing the value for $B$ through successive trials. The procedure continues until $B = C^*$ the optimal solution is found, whereby $C^*$ is the cost of an optimal solution. *DFIDS* combines the advantages of *DFS* (i.e., linear space), and Dijkstra's (i.e., optimality) algorithm [32]. *DFIDS* can prune paths that contain cycles. *DFIDS* exhaustively expands all nodes of the state-space until the current depth is reached. If *DFIDS* fails to find a solution at a particular depth, it must redundantly generate and expand these nodes again in order to expand the nodes at the successive depth. However, the cost of this redundancy is not asymptotically significant with regards to the overall complexity as it is only a linear factor. In other words, the majority of the computation is performed at the deepest level of the search.

## 2.2 Advanced Planning Strategies

### 2.2.1 Heuristics

For a given planning problem $P$, the state-space is exponential in size (given by the number of atoms in $P$). Given the considerable level of complexity involved in finding a solution, uninformed search becomes infeasible. Over time, developments have been made with the aim of exploiting information from the given problem in order to expedite the search. Recall that these strategies are known as *heuristics*. Heuristic techniques offer a strategy for action selection during search to speed up the process of finding a plan. Specifically, the states explored during search can be prioritized according to their estimated distance to the nearest goal state. In order to guide the search towards the goal effectively, an accurate heuristic estimate must be computed efficiently. Furthermore, in order for a planner to be generally applicable on multiple domains, the heuristic must be generated automatically from the given problem [53]. Thus to speed up the search, a

heuristic evaluation function *h(n)* is used to estimate the distance from a node *n* to the goal state. Ideally, the heuristic function should compute the estimate efficiently, and the estimate should be accurate relative to the optimal cost required to achieve the goal from a given node.

**Delete-relaxation Heuristic**

Domain-independent heuristics can be derived from a *relaxed* version of the problem. Relaxations are simplified versions of the original problem and are exploited to produce (admissible) heuristic estimates to further guide the search. The most common relaxation method is the *delete-relaxation* which results in a domain-independent heuristic [18]. This relaxation technique was shown to be successful in previous planning competitions. Furthermore, the delete-relaxation allows for heuristics to be computed quickly and inexpensively.

The delete-relaxation produces a simplified planning problem $P^+$ = <*F*, $O^+$, *I*, *G*> given the original planning problem *P* = <*F*, *O*, *I*, *G*> by maintaining all atoms that were achieved previously. That is, upon applying an action *a* in the delete-relaxation, the positive effects of *a* are added as in *P*, however, the negative effects of *a* are never deleted. This simplification modifies the complexity of computing heuristics from PSPACE-Complete to NP-Complete [1], although determining an optimal solution in the relaxed version remains to be a challenging problem. Hence, the relaxed version of the problem is used to estimate an approximate solution in order to further reduce the complexity to polynomial time.

Upon satisfying the goal state by applying the delete-relaxation, the heuristic estimate can be computed. One of the earliest heuristics developed for domain-independent planning is derived from the delete-relaxation. This strategy can be used to compute one of two estimates, known as the additive $h_{add}(s)$ and max $h_{max}(s)$ heuristics which represent the cost of achieving the goal from state *s*. Note that the computation does not require the extraction of relaxed plans.

Both $h_{add}(s)$ and $h_{max}(s)$ approximate the optimal cost by assuming that each goal of the goal state is achieved independently. Both approaches operate similarly by constructing a graph that estimates the cost of achieving each literal $l_i$ in *s*. Initially, the literals in state *s* are assigned the cost of zero. Subsequently, each action *a* that achieves $l_i$ as a positive effect is added to the graph. In turn, the cost of performing each action is combined with the cost of achieving its preconditions. In contrast, the estimations differ in the method of combining the cost of the action's preconditions.

The $h_{add}(s)$ estimates the cost by summing over all preconditions. Whereas, $h_{max}(s)$ estimates the cost by the most costly precondition.

To formally and conveniently define these heuristics, a new dummy action *End* with a cost of zero is introduced. Its preconditions $g_1...g_n$ are the goals of the problem (i.e., $g \in G$). The effect of the dummy action is a dummy atomic goal *G*. Hence, the heuristic *h(s)* is the function which estimates the cost of achieving this dummy goal *G* from the given state *s*. The heuristic function for $h_{add}$ is defined as follows:

$$h_{add}(s) \stackrel{\text{def}}{=} h_{add}(Pre(End); s) \tag{2.5}$$

whereby $h_{add}(Pre(End); s)$ is the estimated cost of achieving the preconditions of action *End* from *s* as defined from the following expressions:

$$h_{add}(p; s) \stackrel{\text{def}}{=} \begin{cases} 0 & if\, p \in s \\ min_{a \in O(p)[cost(a) + h_{add}(Pre(a); s)]} & otherwise \end{cases} \tag{2.6}$$

and

$$h_{add}(Pre(a); s) \stackrel{\text{def}}{=} \Sigma_{q \in Pre(a)} h_{add}(q; s). \tag{2.7}$$

Within these expressions, *$h_{add}$(p;s)* represents the estimated cost of achieving the atom *p* from *s*. Additionally, *O(p)* denotes the actions which add *p*, and *$h_{add}$(Pre(a);s)* stands for the estimated cost of achieving preconditions of the actions *a* from *s*. In other words, the additive heuristic estimate of producing an atom from a given state is the sum of the (minimum) costs required to achieve each precondition of the action which produces atom. The heuristic function for $h_{max}$ is defined as follows:

$$h_{max}(s) \stackrel{\text{def}}{=} h_{max}(Pre(End); s) \tag{2.8}$$

$$h_{max}(p; s) \stackrel{\text{def}}{=} \begin{cases} 0 & if\, p \in s \\ min_{a \in O(p)[cost(a) + h_{max}(Pre(a); s)]} & otherwise \end{cases} \tag{2.9}$$

and

$$h_{max}(Pre(a);s) \stackrel{\text{def}}{=} max_{q \in Pre(a)} h_{max}(q;s). \tag{2.10}$$

Both $h_{add}$ and $h_{max}$ heuristic estimations are computed similarly with only a few minor differences. Note that equations 2.8 and 2.9 are identical to equations 2.5 and 2.6 respectively, except that $h_{add}$ recursively applies the additive heuristic and $h_{max}$ recursively applies the max heuristic. Further note that the summation in 2.7 is replaced with *max* in equation 2.10. Whereby, the $h_{max}$ estimate of producing an atom from a given state is the max over the (minimum) costs required to achieve the individual preconditions of the action which produces the atom.

The following example demonstrates the computation of $h_{add}(s)$ and $h_{max}(s)$ for the relaxation of a given problem. This example concerns the simplified procedure of starting a vehicle, whereby the cost of each action is 1. For convenience, the arguments of actions and fluents are not explicitly shown. Note, action terms and fluent terms begin with lower-case, and upper-case letters, respectively. Hence, the relaxed problem $P^+ =< F, O^+, I, G >$ is characterized as follows:

- $F$ = { *HaveFuel, HaveKey, FuelAdded, KeyInIgnition, CarOn*}.
- $O^+$ = {*addFuel, insertKey, startCar*}, whereby:

  *Pre(addFuel)* = {*HaveFuel*}, and
  *Add(addFuel)* = {*FuelAdded*}.

  *Pre(insertKey)* = {*HaveKey*}, and
  *Add(insertKey)* = {*KeyInIgnition*}.

  *Pre(startCar)* = {*FuelAdded, KeyInIgnition*}, and
  *Add(startCar)* = {*CarOn*}.

- $I$ = {*HaveFuel, HaveKey*}.
- $G$ = {*CarOn*}.

Figure 2.1 illustrates the computation for both the additive $h_{add}$ and max $h_{max}$ heuristic estimates from the initial state $S_0$ of the relaxed version of the problem $P^+$.

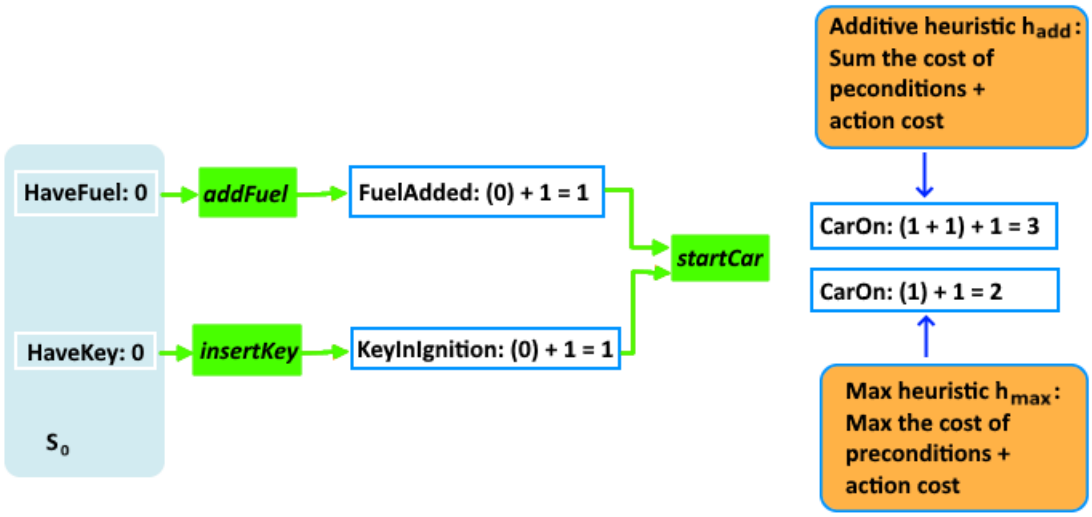



FIGURE 2.1: Visualization of $h_{add}$ and $h_{max}$ estimations

Both estimations can be computed efficiently. Although, $h_{add}$ is not admissible and may overestimate the cost since it does not account for positive interactions among the goals. Additionally, $h_{add}$ may underestimate the cost required to achieve the goal since the cost of achieving multiple atoms may be more than the sum of achieving them separately. However, $h_{add}$ is typically more informative than $h_{max}$ since it considers all goals in the computation. Whereas, $h_{max}$ is admissible since the cost of achieving one sub-goal is always less than or equal to the cost of achieving all sub-goals. However, $h_{max}$ is not as informative since the estimation considers only one of the atoms in each action precondition.

#### Alternative Heuristic Estimation Approaches

In order to produce an optimal plan, admissible heuristics are required. In general, heuristic estimates are computed by applying non-negative operator costs. However, Pommerening et al. [57] demonstrate the advantage of using negative heuristic estimates by applying general operator cost partitioning. Cost Partitioning is a procedure used to make several admissible heuristics additive such that their sum is admissible [30]. In other words, upon computing the heuristic estimate for each sub-goal of the goal state, this technique ensures that the total sum will not overestimate the optimal cost required to satisfy the goal. The rationale behind this work reasons as follows. The total estimate must be admissible given that each heuristic is evaluated by a

reduced cost function, and therefore contributes a factor of the optimal cost. As a result, the heuristic estimate based on general cost partitioning demonstrates strong performance and that the restriction of applying non-negative operator costs is not compulsory.

Admissible heuristics are generally consistent. A heuristic is consistent if and only if the triangle inequality holds, $h(n) \leq c(n, n') + h(n')$ [20]. Meaning that, the heuristic estimate of state $n$ is less than or equal to the cost of the least-cost path from $n$ to $n'$ plus the estimate of the cost remaining $h(n')$ to the goal. Consistency is favoured to prevent issues such as node re-expansion. For instance, without consistent heuristics, a heuristic search algorithm may identify a shorter path to a node that was previously expanded and stored in the CLOSED list. In this case, the node will be placed back into the frontier and expanded once again. Nevertheless, admissible heuristics are not required to be consistent. In turn, Felner et al. [15] explore the advantages of using inconsistent heuristics as they are are not well investigated. Specifically, it is shown that producing effective inconsistent heuristics can benefit the search by reducing the number of node expansions. Furthermore, previous notions regarding the use of inconsistent heuristics are refuted. In particular, inconsistent heuristics are shown to be computed with less effort than previously shown. Furthermore, they are shown to exhibit the worst-case of node re-expansion in exponential time only on artificial graphs. The experimental results produced in this paper demonstrate considerable improvement in performance with inconsistent heuristics for IDA* and A* search algorithms.

### 2.2.2 Advanced Search Algorithms

The following review consists of well-established advanced searching algorithms. These algorithms apply search over weighted directed graphs by featuring heuristics to guide the search effectively [53, 11]:

1. *Best-First Search*:

   Best-first search implements the search frontier as a priority queue. Upon expansion of neighbouring nodes during search, the nodes are assigned a cost given by the evaluation function $f(n) = g(n) + h(n)$. As defined previously, $g(n)$ represents the accumulated cost of achieving the current node $n$ from the initial state. Whereas, $h(n)$ represents the heuristic estimate of achieving the goal state from the current node $n$. Best-first search is able to prune paths and cycles by prioritizing nodes that have the lowest estimated distance to the goal.

However, it suffers from the disadvantage of using exponential time and space $O(b^d)$ where $b$ is the branching factor and $d$ is the length of the optimal solution.

2. *Greedy Best-First Search (GBFS)*:

   *GBFS* is a best-first search that uses the evaluation function $f(n) = h(n)$, where $h(n)$ is the estimated cost from the current node $n$ to the goal. *GBFS* is known to be one of the most popular forms of search used for satisficing planning, whereby the solution found is not required to be optimal. In turn, *GBFS* will find a solution in less time than best-first search given an accurate heuristic function. However, given that *GBFS* solely relies on the heuristic estimate to guide the search, it does not guarantee that an optimal solution will be found.

3. *Recursive Best-First Search (RBFS)*:

   *RBFS* is a type of best-first search that maintains the current node with the least cost along with its ancestors and their immediate siblings on the recursion stack. During search, if a node with a higher cost to the current node is encountered, the algorithm recursively returns to search through a new path that stems from their deepest common ancestor. Irrespective of the cost function used, *RBFS* uses linear space according to the length of the optimal solution.

4. *A* Search (A*)*:

   *A** [20] is an instance of best-first search that uses the the evaluation function $f(n) = g(n) + h(n)$. In turn, the heuristic function estimates the minimum total cost of any solution path that involves node $n$. *A** produces the optimal plan when the heuristic function is admissible. The main disadvantage of *A** is the use of exponential space, which in turn exhausts available memory for exceedingly difficult problems.

5. *Memory-Bounded A* Search (MA*)*:

   *MA** search is an extension to *A** whereby they share the same evaluation function $f(n) = g(n) + h(n)$. However, *MA** addresses the memory limitation exhibited by *A** [71]. Briefly, the reduction in memory usage is achieved by removing the least promising nodes from the frontier when a prescribed bound on memory is reached. In addition to moderating the limitation on memory, *MA** can identify more promising paths to nodes that were previously expanded. As a result, *MA** is able to improve the performance of *A** on exceedingly difficult problems that possesses combinatorially many paths from the initial node.

6. *Depth-First Iteratively Deepening A* Search (IDA*)*:

   *IDA** [32] is a form of *DFIDS* which relies on the evaluation function used by *A**. In order to alleviate the exponential memory limitation of *A**, *IDA** does not maintain the list of previously generated nodes. As in *DFIDS*, *IDA** prunes paths that contain cycles and paths that exceed the prescribed bound *B*. In addition, the search prioritizes nodes by giving precedence to the node with the lowest heuristic estimate (i.e., closest to the goal). Although, *IDA** is not asymptotically optimal [42]. Meaning that, it does not perform better (i.e., in terms of a solution quality, time or space complexity) than an optimal algorithm. This is mainly caused by the re-expansion of nodes since *IDA** does not maintain the list of preciously generated nodes.

7. *Depth-First Branch-and-Bound Search (DFBnBS)*:

   *DFBnBS* [37] is a variation of *DFS* that makes use of heuristic information while maintaining the space efficiently. Once a sub-optimal solution is found, its cost serves as an upper bound for the optimal solution. This strategy improves pruning efficiency by excluding paths which exceed the cost of the current solution found. In addition, *DFBnB* never expands any node more than once (unlike IDA*). As a result, a solution can be found quickly, upon which *DFBnBS* will continue to improve it in order to return an optimal solution.

8. *A* with Lookahead (AL*)*:

   *AL** [66] is another extension of *A** that takes advantage of both best-first search and depth-first search. *AL** significantly reduces memory requirements and the time required to find a solution for several domains relative to *A**. Specifically, upon expanding the search frontier of best-first search, *AL** performs local look-aheads through *DFS*. Moreover, the *DFS* strategy used is the depth-first branch-and-bound algorithm in order to guarantee an optimal solution is found.

9. *Weighted A* Search (WA*)*:

   *WA** is a form of *A** search that applies a constant $W > 1$ to place more emphasis on the heuristic estimate. Hence, the evaluation function used by *WA** is $f(n) = g(n) + W \cdot h(N)$, whereby $W \cdot h(n)$ is the scaled heuristic estimate. In general *WA** finds a solution in less time than *A**. However, the heuristic evaluation $W \cdot h(n)$ no longer produces an admissible estimate even if the heuristic $h(n)$ is admissible.

10. *Anytime Weighted A* Search (AA*)*:

*AA** is an extension to *WA** search. Hansen and Zhou [19] developed improvements on the *WA** algorithm that enables a suboptimal solution to be found quickly. Subsequently, the search improves upon the quality of the solution by imposing two bounds in order to facilitate the convergence of the sequence of solutions to the optimal solution. The lower bound consists of the admissible heuristic evaluation function $f(n) = g(n) + h(n)$. Whereas, the upper bound consists of the inadmissible heuristic evaluation function $f(n) = g(n) + W\ h(n)$ as in *WA**. In turn *AA** will find a solution in less time than *A**, however at the cost of the solution quality until convergence occurs.

11. *Restarting Weighted A* Search (RWA*)*:

*RWA** search [63] expands upon *WA** search by decreasing the scalar weight $W$ used within the evaluation function throughout search. Furthermore, upon identifying a new suboptimal solution, *RWA** restarts the procedure from the initial state. These modifications demonstrate an improvement in time efficiency, especially when the heuristic estimate used is not as informative.

12. *Real-Time A* Search (RTA*)*:

*RTA** [33] is a form of *A** that uses a similar evaluation function $f(n) = g(n) + h(n)$. However, instead of representing the cost from the initial state to a node $n$, $g(n)$ represents the cost of the edge from a neighbouring node to the current node $n$. A common occurrence exhibited by *A** is repeated backtracking given that the most optimal move is to return to previous state. Instead, *RTA** performs backtracking when the estimated distance of the current path $f(n)$ exceeds the sum of: 1) the cost of returning to a previous state $g(n)$, and 2) the estimated cost of reaching the goal state from the current node $h(n)$. As a result, the search explores the neighbour with minimum cost, as it maintains the previous state with the best cost from the neighbouring nodes. As a result, *RTA** can make use of information achieved from previous searches while planning.

## 2.3 State-of-the-Art Planning

This section reviews the conventions used by modern state-of-the-art planning. These standards provide a way in which planners can be evaluated against each other. Included is an overview concerning the standard representations of planning problems. These languages and their variations may support distinct as well as overlapping features required to accurately express a variety problem domains. Lastly, we briefly review some foundational works developed by the planning community that has influenced many of the modern approaches in state-of-the-art planning.

### 2.3.1 Planning Domain Descriptions

Given the logical structure of given problems and the various methods in which search can be performed, an expressive language is required to describe planning problems so that modern planning can be applied. This section outlines the language which has been predominantly used in classical planning, known as STRIPS [16]. We then present variants and extensions of the STRIPS language that are favoured by the AI planning community.

STRIPS (Stanford Research Institute Problem Solver) developed by Fikes and Nilsson in 1971 [16] is a well known language used by modern planners to describe planning tasks. STRIPS uses first-order logic notation to describe objects in a given environment and the ways in which they can be manipulated. Specifically, the language represents states as conjunctions in FOL and actions by their preconditions and effects (i.e., add and delete lists). A planning problem expressed in STRIPS is defined by a tuple
<*F, O, I, G*>:

- *F*: is the set of propositions,
- *O*: is the set of operators (i.e., actions)
- *I*: $I \subseteq F$ is the set of atoms which are initially true, whereby atoms not in *I* are all assumed to be false (i.e., closed-world assumption is used),
- *G*: $G \subseteq F$ is the set of atoms that compose the goal state.

Furthermore, each action *o* in the operator set *O* is defined using another tuple <*Pre, Add, Del*>:

- *Pre(o)*: is the list of atoms that must be true as a requirement in order to execute action *o*,
- *Add(o)*: is the list of atoms that become true as a result of executing action *o*,
- *Del(o)*: is the list of atoms that become false as a result of executing action *o*.

Further development of STRIPS motivated the progress of several additional formalisms. With assistance from the Planning Competition committee (1998), Drew McDermott developed the Planning Domain Definition Language (PDDL) [52]. Predominantly, PDDL is used as the standard to express domains and problems by the AI planning community. For instance, the International Planning Competition uses PDDL to represent all instances of their benchmarks. PDDL was mainly inspired by STRIPS and Pendault's Action Description Language (ADL) [54] in addition to other variations. The development of these languages led to the addition of more complex features that in turn enabled the representation of exceedingly challenging domains. For example, the following briefly outlines some of the distinguishing features of STRIPS and ADL [64]:

- *STRIPS*: The STRIPS language only allows for positive literals within states. STRIPS operates under the closed-world assumption. Meaning, all literals which are not mentioned in the state are considered false. The goal state can only be composed of conjunctions that consist of ground literals. Additionally, effects of actions can only consist of conjunctions of positive literals. STRIPS does not provide support for equality, types, derived predicates or conditional effects.
- *ADL*: The ADL language allows for both positive and negative literals within states. Unlike STRIPS, ADL operates under an open-world assumption. Meaning, information regarding all literals which are not mentioned in the state is unknown. The goal state may consist of conjunctions or disjunctions that involve ground literals as well as literals that include quantified variables. ADL supports effects of actions that may be context-dependent. ADL provides support for equality and types.

Proceeding with PDDL, the language extends STRIPS and ADL with additional features. PDDL supports the use of derived predicates as well as numeric-valued fluents. Furthermore, PDDL

supports durative actions by providing an explicit representation for time. For our purposes, the initial state and goal state are composed of a conjunction of literals. Also, all actions are explicitly defined by their preconditions and effects, whereby effects are not context-dependent. The conditions and effects are composed of logical formulas expressed by the given predicates of the domain and logical connectives.

The syntax for PDDL is similar to Lisp. Atomic formulas in PDDL have the form *(functor arguments)*. Actions are expressed as terms that contain variable arguments (preceded by a question mark). Each action has a set of preconditions which must be satisfied in the current situation for the action to be applicable. Additionally, each action may have several effects that occur in the next situation once the action is performed. As a result, a literal effect may become true or become false in the next situation.

Figure 2.2 illustrates the PDDL definition for the action *pick-up* within the *blocksWorld* domain. Briefly, *blocksWorld* is a well known domain used in classical planning. *BlocksWorld* consists of a set of blocks, a single table, and a single robotic arm which may reposition the blocks (one at a time) in the environment. Each block can only be directly on top of another block or directly on the table. The blocks can be arranged successively to construct towers. The planning task involves using the arm to manipulate the blocks from their initial position in order to produce the desired configuration of blocks. Examples of this domain are used frequently to illustrate and clarify the concepts throughout this thesis.

```
(:action pick-up
        :parameters (?x)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
        (and (not (ontable ?x))
        (not (clear ?x))
        (not (handempty))
        (holding ?x)))
```

FIGURE 2.2: Displays the PDDL definition for action *pick-up* within the *blocksWorld* domain

As shown in Figure 2.2, the definition for the action *pick-up* states that it is possible to pick up a block *?x* if the block is clear (i.e., no other block is on its top), the block is on the table, and the hand is empty. As a result of picking up a block, it is no longer on the table, it is not clear, and

the hand is not empty. Additionally, the block is being held. This formalism is able to provide a complete description of the resulting situation upon executing actions ($A_1$,...,$A_n$) given the initial situation $S_0$.

Planning problems expressed in PDDL are given by two descriptions. These are the *domain description* and *problem description*. Included in the domain description, are the object types, predicates and definitions of actions for the given problem domain. The problem description includes the objects, initial state, and goal state which are specific to the given problem instance. Combined, the domain description and problem description together exhibit an instance of the planning problem.

### 2.3.2 State-of-the-Art Planners

The approaches taken by modern state-of-the-art algorithms to perform classical planning range considerably. Generally, these algorithms apply path-finding search over weighted directed graphs to determine a solution for the given planning task. These algorithms vary in the direction in which state-space search is performed. Furthermore, algorithms may conduct plan-space search, whereby search is performed in a space of partial plans. These (incomplete) plans consist of action sequences that are not necessarily ordered. Additionally, algorithms can exploit data structures known as planning graphs by extracting accurate heuristics to direct the plan-graph search [2, 3]. These strategies have been developed over time by the researchers involved within the AI planning community. This section briefly summarizes some of the foundational and noteworthy works that lead to modern approaches in planning.

Early planners, such as *GPS* developed by A. Newell and H. Simon in the 1960s [14], commonly applied search in the forward direction. Although, without the use appropriate heuristics, this strategy was largely impractical. In particular, forward search is likely to consider irrelevant actions, which renders the search infeasible especially for problems with a large number of objects. Among the first planners to apply domain-independent heuristics, *Unpop* was introduced by Drew McDermott in 1996 [50, 51]. Unpop is a forward search planner whereby the search is directed by a regression-based heuristic. Regression is applied on the goal through (ground) actions without accounting for goal interaction. The heuristic computation is re-applied at every step in forward search until the goal is reached. *Unpop* exhibited competitive performance against other

planners at the time of its introduction. Afterwards, planners adopted similar and more advanced use of heuristics to improve results and solve more challenging problems.

In 1997, GRAPHPLAN was established by Blum and Furst [3]. GRAPHPLAN operates by alternating between two phases. The first phase performs forward search. The search frontier is expanded from the initial state towards the goal state on a best-first basis. In order to direct the search, the second phase computes the heuristic estimate by performing plan-graph search. Briefly, the planning graph is generated and extended from the current state towards the goal. Once the goal state is satisfied by the relaxed data structure, the relaxed plan is extracted through a back-chaining algorithm. As a result, the reachability heuristic is computed as the cost of the relaxed plan. The heuristic is inexpensive to extract and accurately estimates the distance between states. Moreover, the estimate is shown to be admissible and more accurate than the additive or max heuristic produced by delete-relaxations [5]. Although, the heuristic is recomputed at each step of forward search. As a result, GRAPHPLAN significantly outperformed popular planners at the time (i.e., *Prodigy* and *UCPOP*). Mainly, the algorithm excelled by significantly pruning a sizeable portion of the state-space with the use of mutual exclusion relations (explained in subsection 2.5.5). Due to its exceptional performance, GRAPHPLAN was established to be among the most ideal methods of computing the reachability information [5].

The *Heuristic Search Planner* (HSP) developed by Bonet and Geffner in 1998 [4] is a forward search, forward heuristic construction planner. *HSP* demonstrated competitive results against GRAPHPLAN in the Artificial Intelligence Planning and Scheduling (AIPS) Planning Competition of 1998. *HSP* relies on a *hill-climbing* strategy. In particular, the planner takes advantage of the $h_{add}$ heuristic, computed by the delete-relaxation technique, to guide hill-climbing search. Whereby, hill-climbing selects a frontier node that minimizes the $h_{add}$ estimate. Additionally, the strategy is extended by conducting limited plateau search and restarts. Meaning, that if the search exceeds a prescribed number of consecutive states which do not decrease the $h_{add}$ estimate (i.e., plateau), then the search is restarted. Moreover, visited states are maintained in a hash table so that they are avoided in subsequent iterations. Also, the search is restarted from different states to avoid similar paths traversed in previous iterations. As a result, *HSP* was able to solve more problems than other competing planners. However, it frequently produced sub-optimal plans or took more time relatively.

The *HSPr* (*r* for regression) planner [4] applies backward search with forward heuristic construction. Given the limitation of re-computing the heuristic at each step in forward search, *HSPr* constructs the heuristic only once. This technique was originated by the Greedy Regression Table (GRT) System [60]. Although computing the heuristic only once saves a considerable amount of time and computational effort, it has some limitations. For instance, *HSPr* exhibits the issue of encountering invalid states in the regression space since planning is performed from partial goal states. To manage this issue, mutual exclusion relations are computed such that each state in regression space is assessed for violation of these relations. However, this additional computational effort uses a considerable amount of time and memory. As a result, *HSPr* did not perform more significantly with respect to *HSP*.

The *Fast-Forward* (FF) planner was established by J. Hoffman in 2001 [24, 25]. *FF*, as in *HSP*, constructs the heuristic and searches in the forward direction. However, *FF* uses *GRAPHPLAN* to compute the heuristic instead of $h_{add}$. Furthermore, *FF* does not assume goals are achieved independently as in *HSP*, since this assumption does not account for positive interactions. *FF* constructs a planning graph from the current state towards the goal by using relaxed actions. The relaxed plan is extracted from the graph, whereby the heuristic estimate is computed to be the length of the plan. This estimate is more informative than the additive heuristic. As a result, the combination of techniques used by *FF* was shown to outperform all planners including *HSP* at the AIPS-2000 competition.

The *Fast-Downward* (FD) planner [23], introduced by Malte Helmert, has been in development since 2003. *FD* performs heuristic forward search similar to other planning systems such as *HSP* and *FF*. Although unlike other planners, *FD* does not directly use the propositional PDDL representation of a planning task. Rather, *FD* reformulates the PDDL representation into *multi-valued planning tasks*. The modified representation allows the structure in the description to be leveraged by making many of the implicit constraints of a propositional planning task explicit. As a result, *FD* is able to compute its heuristic function, namely *causal graph heuristic*, by solving the task in a hierarchical fashion. Specifically, the planner begins from the top-level goals and recursively operates down the causal graph until the remaining subproblems become simplified graph search tasks. In turn, *FD* was the top performing planner for the *classical track* of the IPC-2004. *FD* was shown to be competitive across domains against the state-of-the-art planners involved

in the competition. For certain domains, *FD* outperformed all other planners. Although, other domains highlight limitations exhibited by *FD*. Specifically, causal graphs generated for domains that possess *blocksWorld*-like sub-problems become significantly dense and causes *FD* difficulty in the recursive procedure. Nevertheless *FD* and extensions of *FD* have consistently shown to perform well in ensuing competitions.

### 2.3.3 Contribution to State-of-the-Art Planning

Subsequent planning competitions have given rise to much development in state-of-the-art planning. In turn, each competition involves more challenging domains that further highlight the limitations of competing planners. With regards to the most recent competition, IPC-2018, a strong emphasis was placed on the drawbacks of grounding. Recall that prior to performing search, modern planners apply an instantiation procedure to produce a grounded transition system. Subsequently, the planners can apply advanced search algorithms to determine solutions to a given planning task. Modern planners have significantly improved their techniques with regards to constructing and pruning the state-space for efficient search. However, the limitations of grounding are unavoidable for exceedingly challenging problems.

Our team participated in the IPC-2018 by submitting the *Organic Chemistry Synthesis* benchmark. Briefly, this benchmark consists of problems relating to organic synthesis in chemistry and was originally developed by Heifets and Jurisica [22, 21]. The benchmark set was then amended by our colleagues, namely Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski [45, 44] to be used in domain-independent planning. Whereby, we have corrected the previous encoding (summer 2016) by consulting with professors in chemistry at Ryerson University, Dr. Anne Johnson, Dr. Russell Viirre, Dr. Sharonna Greenberg, and Dr. Andrew McWilliams. The problems range in difficulty by the number of objects involved. Additionally, certain problems require lengthy and elaborate solutions. Moreover, most action terms involve a significant number of arguments given the extensive number of preconditions that are required by each action.

This benchmark was used to gauge the performance of competing planners across the *satisficing* and *agile* tracks of the competition. The specifications limit the planners to use a single CPU core and 8GB of memory. The satisficing track requires planners to produce a valid plan that is not necessarily optimal. For each problem, the metric used to gauge the performance of a planner

is: ($C^*$ / $C$) where $C^*$ is the length of the optimal solution and $C$ is the length of the cheapest plan found. Otherwise, a planner receives a score of 0 if no solution is found. As such, higher scores are given to planners that produced sub-optimal plans with higher quality. The agile track concerns the CPU time taken to determine a valid plan, whereby a time limit of 5 minutes is imposed. The quality of the plan is overlooked. A planner is given: a score of 1 if it solves a problem in less than 1 second, 0 if no solution is determined, or a factor $0 < k < 1$ otherwise. This factor is computed as $k = 1 - ( log(T) / log(300) )$, given $T$ as the time (seconds) taken to solve the task. For a particular domain, the total score of a planner is the sum of its scores for all task in that domain.

Results of the competition were presented at the 28th International Conference on Automated Planning and Scheduling (ICAPS-2018). For our domain, the results illustrate a lack in significant performance across planners in general. Specifically, the top score awarded in the satisficing track was 12 out of 20 for our benchmark as shown in Figure 2.3. Whereas, the top satisficing score awarded to planners on all other domains was higher (aside from the *termes* domain). This indicates that more problems could be solved, or solved more optimally for other domains than the organic synthesis domain.

Furthermore, the highest score awarded in the agile track was 4.8 out of 20 as shown in Figure 2.4. Whereas, the top agile score awarded to planners on all other domains was higher (except for the *agricola* domain). This result also indicates that more problems could be solved, or solved much faster for almost all other domains than our benchmark.

As a result, we successfully highlight the limitations of grounding exhibited by modern planners through this set of problems. For our effort, we received the *Outstanding Domain Submission Award* awarded by the organizers, Florian Pommerening and Alvaro Torralba. Our benchmark is discussed in further detail in Section 3.5.

| Sat score | agricola | caldera | data-net. | flashfill | nurikabe | org.-syn. | settlers | snake | spider | termes | SUM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Stone Soup | 13 | 14 | 10 | 13 | 18 | 9 | **16** | 7 | 10 | 8 | 123 |
| Remix | 13 | 14 | 10 | 12 | **18** | 9 | 16 | 7 | 10 | 6 | 120 |
| DUAL-BFWS | 12 | **17** | 11 | **16** | 14 | 11 | 6 | 9 | 12 | 5 | 119 |
| Saarplan | **14** | 11 | 12 | 13 | 16 | 11 | 9 | 8 | 10 | 7 | 116 |
| DecStar | 12 | 13 | 10 | 13 | 12 | 9 | 15 | 4 | 12 | 6 | 111 |
| Cerberus | 10 | 10 | 11 | 9 | 15 | **12** | 9 | 5 | 13 | 7 | 108 |
| LAMA 2011 | 9 | 13 | 7 | 13 | 10 | **12** | 15 | 3 | 13 | 7 | 107 |
| BFWS-Pref. | 11 | 15 | 8 | 11 | 12 | 7 | 8 | **15** | 10 | 5 | 106 |
| Cerberus-gl | 10 | 10 | 11 | 9 | 15 | **12** | 9 | 5 | **14** | 6 | 106 |
| OLCFF | 13 | 11 | 12 | 0 | 17 | 9 | 0 | 7 | 11 | 7 | 92 |
| POLY-BFWS | 13 | 17 | 11 | 8 | 10 | 5 | 9 | 2 | 8 | 2 | 90 |
| IBaCoP | 10 | 5 | **14** | 0 | 6 | 8 | 0 | 8 | 8 | **10** | 73 |
| IBaCoP2 | 11 | 6 | 11 | 0 | 7 | 8 | 0 | 7 | 7 | 7 | 66 |
| MERWIN | 10 | 0 | 10 | 0 | 5 | **12** | 0 | 4 | 11 | 7 | 62 |
| mercury | 12 | 0 | 8 | 0 | 5 | 11 | 0 | 3 | 12 | 7 | 61 |
| DFS+ | 10 | 12 | 6 | 1 | 5 | 5 | 7 | 4 | 7 | 0 | 60 |
| fs-sim | 11 | 6 | 5 | 0 | 10 | 4 | 0 | 7 | 4 | 3 | 53 |
| fs-blind | 3 | 6 | 5 | 0 | 12 | 4 | 0 | 7 | 5 | 7 | 50 |
| freelunch-dr | 8 | 1 | 0 | 0 | 0 | 6 | 0 | 5 | 0 | 0 | 22 |
| freelunch-ma | 0 | 2 | 2 | 0 | 3 | 8 | 0 | 1 | 0 | 0 | 16 |
| Symple-2 | 1 | 2 | 0 | 0 | 2 | 5 | 0 | 0 | 0 | 1 | 11 |
| Symple-1 | 1 | 2 | 0 | 0 | 2 | 5 | 0 | 0 | 0 | 1 | 11 |
| alien | 4 | 0 | 1 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 9 |

FIGURE 2.3: Experimental results for the satisficing track produced by planners submitted to the IPC-2018

| Agl score | agricola | caldera | data-net. | flashfill | nurikabe | org.-syn. | settlers | snake | spider | termes | SUM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BFWS-Pref. | 2.3 | 6.9 | 6.1 | **8.8** | **7.5** | 3.9 | 2.4 | 8.9 | **5.6** | 3.8 | 56.1 |
| LAMA 2011 | 0.8 | 6.6 | 7.6 | 7.4 | 6.3 | 2.9 | **7.1** | 2.0 | 4.6 | **7.6** | 52.7 |
| Saarplan | 1.4 | 6.6 | **9.5** | 3.4 | 7.5 | 1.9 | 3.8 | 3.8 | 2.0 | 6.3 | 46.3 |
| DUAL-BFWS | 1.6 | **7.6** | 4.4 | 8.0 | 7.1 | **4.8** | 1.7 | 3.8 | 4.2 | 3.1 | 46.2 |
| Remix | 1.2 | 6.1 | 6.6 | 6.0 | 7.1 | 3.3 | 5.6 | 1.6 | 1.5 | 5.4 | 44.3 |
| POLY-BFWS | 2.2 | 7.5 | 5.4 | 6.7 | 7.4 | 2.8 | 1.8 | 2.5 | 4.8 | 1.9 | 43.0 |
| DecStar | 1.4 | 5.8 | 5.3 | 3.9 | 6.4 | 2.3 | 5.6 | 1.8 | 2.6 | 6.3 | 41.4 |
| OLCFF | 1.3 | 6.6 | 9.1 | 0.4 | 7.4 | 1.7 | 0.0 | 3.8 | 1.7 | 6.0 | 38.1 |
| Cerberus | 0.5 | 5.9 | 4.8 | 2.4 | 7.4 | 1.5 | 1.7 | 2.7 | 0.7 | 6.8 | 34.4 |
| Cerberus-gl | 0.4 | 5.8 | 4.8 | 2.4 | 7.5 | 1.6 | 1.7 | 2.6 | 0.7 | 3.6 | 31.0 |
| LAPKT-DFS+ | 2.4 | 6.6 | 1.9 | 0.3 | 4.1 | 2.0 | 2.6 | 0.7 | 3.5 | 0.0 | 24.1 |
| mercury2014 | 1.1 | 0.0 | 7.1 | 0.0 | 1.1 | 2.5 | 0.0 | 1.9 | 3.4 | 6.4 | 23.5 |
| fs-blind | 0.5 | 3.4 | 2.4 | 0.0 | 7.4 | 0.2 | 0.0 | 4.7 | 1.5 | 3.4 | 23.5 |
| fs-sim | **2.5** | 3.3 | 3.2 | 0.0 | 6.5 | 0.4 | 0.0 | 2.7 | 1.1 | 3.0 | 22.8 |
| MERWIN | 0.9 | 0.0 | 7.0 | 0.0 | 1.1 | 2.5 | 0.0 | 1.8 | 2.8 | 5.7 | 21.7 |
| freelunch-dr | 1.1 | 0.9 | 3.4 | 0.0 | 0.0 | 1.2 | 0.0 | **10.8** | 1.8 | 0.0 | 19.2 |
| IBaCoP | 0.3 | 0.4 | 1.5 | 0.0 | 0.1 | 1.0 | 0.0 | 0.4 | 0.0 | 0.8 | 4.5 |
| freelunch-ma | 0.0 | 1.4 | 0.8 | 0.0 | 0.6 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.9 |
| alien | 0.6 | 0.0 | 1.1 | 0.0 | 0.0 | 1.8 | 0.0 | 0.0 | 0.0 | 0.0 | 3.5 |
| IBaCoP2 | 0.3 | 0.1 | 1.1 | 0.0 | 0.0 | 0.5 | 0.0 | 0.4 | 0.0 | 0.6 | 3.0 |
| Symple-2 | 0.0 | 0.1 | 0.0 | 0.0 | 0.4 | 1.7 | 0.0 | 0.0 | 0.0 | 0.0 | 2.1 |
| Symple-1 | 0.0 | 0.1 | 0.0 | 0.0 | 0.5 | 1.4 | 0.0 | 0.0 | 0.0 | 0.0 | 2.1 |

FIGURE 2.4: Experimental results for the agile track produced by planners submitted to the IPC-2018

## 2.4 Regression

Classical planning approaches have significantly progressed with regards to the search algorithms they employ. For instance, these methods vary in the direction of search. Search can be performed in the forward direction by progressively applying actions from the initial state towards the goal until the goal state is satisfied. In contrast, backward search is performed from the (incompletely specified) goal state towards the initial state by regressing through the actions until all sub-goals (i.e,. preconditions of actions) are satisfied by the initial state. Forward search is known as *progression*, whereas backward search is known as *regression*. Additionally, an algorithms may use a combination of both forward and backward search to perform *bidirectional search* [26] until a shared state is produced by both search frontiers.

Mainly, progression and regression are the two search strategies mostly explored, whereby progression is predominantly used in planning [64]. The majority of modern planners construct heuristics and perform search in the forward direction. This trend is mainly due to the lack of thorough research involving regression to produce informative heuristics. Furthermore, progression-based planners have been shown to outperform any planner that operates using regression currently [18]. Given that the issues posed by regression are not as extensively studied, a significant amount of our research effort is dedicated to examining regression-based search and related strategies.

Richard J. Waldinger first introduced the notion of goal-regression in 1975 for classical AI planning [43, 70]. However, prior to its development in AI planning, regression techniques have been shown to be effective when solving difficult tasks in other fields as well as in general problem-solving. In particular, the *retrosynthetic* approach proposed by Elias. J. Corey [10, 9] in 1969 made use of a backward strategy to solve and analyse problems in the field of organic synthesis. Specifically, given a desired target molecule as the goal, retrosynthesis was applied to transform the goal molecule into preceding sub-goal molecules which are required to produce the target. The process is repeated until the resulting sub-goal structures are available initially.

In terms of general problem-solving, we as human beings commonly use the goal-regression approach. For example, consider the goal of attending a conference being held in London, England. This goal can be decomposed into several sub-goals such as, scheduling an airline ticket,

arranging for transportation to the airport, and booking accommodations for the time spent in the city. These sub-goals can conveniently be satisfied by working backwards. Specifically, an individual will first consider the date and time when the conference will take place before arranging the reservations for a hotel. Subsequently, the flight ticket is booked, then arrangements for transportation to the airport are made. Working backwards simplifies the decision-making process and is especially ideal for situations that enable a large number of choices when planning in the forward direction.

Regression operates to produce a plan required for achieving the goal by searching backwards from the goal state towards the initial state. In general, for a given goal $G$, regression proceeds to find an action $A$ that will make $G$ true upon execution. Regression continues by finding new sub-goals $G'$, such that if $G'$ is required for the application of action $A$, then $G$ will become true as a result. Identifying new sub-goals $G'$ is the process known as *regressing goal G through action A*. Since the set of atoms within the goal state is to be achieved, the final action must contribute some atom(s) and not delete any atoms present within the goal state. It is possible that the last action deletes extra atoms present in the state to achieve the goal state, or replaces some extra redundant atoms with the atoms that must be present in the goal state. Hence, the set of atoms required to apply the final action $A$ (i.e., $G' = Pre(A)$), as well as atoms within the goal that are not added by $A$ are necessary to achieve $G$. In turn, this procedure produces the state-space defined as the *regression state model* [18].

With regards to planning, regression provides a method of solving tasks which are especially difficult to solve by progression. For instance, an initial state's size may theoretically be unlimited, whereas the goal state is generally compact and small (i.e., as in organic chemistry synthesis). In turn, regression can determine a plan by constructively identifying the *relevant* actions at each stage during the procedure. Another advantage of regression is the conservation of time with regards to the heuristic computation. Since regression uniquely achieves the initial state from the goal state, its necessary to be computed only once if the information obtained from regression is maintained. In turn, the re-computation of heuristic estimates exhibited by forward search can be avoided. These advantages further motivated our interest to investigate search strategies involving regression because it demonstrates potential in solving difficult problems.

Although, regression provides a number of considerable advantages, it suffers from several limitations. For example, given that the goals and sub-goals are represented as conjunctions, regression is performed on each conjunct separately, then the resulting subplans are joined to produce the final plan. Producing a valid plan is difficult given that the subplans may contain actions which delete effects produced by other actions that contribute to the goal. Furthermore, when regressing a goal $g$, given a considerable number of actions that achieve $g$ are found, then the procedure spends a significant amount of effort to satisfy $g$ that is similar to performing forward search. In this thesis, we recognize these issues and have developed techniques in an attempt to alleviate them within our work as discussed in Chapter 4.

The *regression space R(P)* modelled as a STRIPS planning problem *P = (F, I, O, G)* differs from the *progression state model* as follows:

- *F*: represents the set of states *s* as sets of atoms,
- *G*: is the initial state $S_0$
- *I*: denotes the set of goal states *s* for which $s \subseteq I$
- *O*: is the set of actions *A(s)* that can be applied in *s* that are:

  i) *relevant* whereby at least one positive effect of *a* is included in *s* (i.e., $Add(a) \cap s \neq 0$), and

  ii) *consistent* whereby the negative effects of *a* are not included in *s* (i.e., $Del(a) \cap s = 0$).
- *f(a,s)*: is the transition function that defines the resulting state $s' = (s \setminus Add(a)) \cup Pre(a)$ as a consequence of applying action $a \in A(s)$, and
- *c(a,s)*: are the action costs

As in the *progression state model*, the solution in *R(P)* consists of a sequence of actions that achieves the goal state from the initial state but in reverse order. The two state models have a number of differences. Specifically, the regression space defines the initial state as the set of propositions that compose the original goal state. Correspondingly, the goal state is defined as the set of propositions that compose the original initial state. Although the states in both models are represented by a set of atoms, their meanings are quite different. In the progression space, the

states represent *complete truth assignments* under a closed-world assumption. In other words, every atom that is not explicitly stated in the initial state of the progression space *S(P)* is considered false. Conversely, the states in the regression space *R(P)* are represented by *partial-truth assignments*. Thus, information regarding the atoms which are not explicitly stated in *R(P)* is unknown. In turn, the procedure may expend valuable resources by producing dead-end states that cannot be reached by any plan from the initial situation [4]. Instead, states in *R(P)* are considered as goals and subgoals whereby actions are applied in reverse order. Even so, a plan produced from the regression space can be shown to be both *sound* and *complete*, meaning that a valid solution will be found (if one exists) [56].

### 2.4.1 Regression Formalism

This section reviews the regression mechanism more formally. In detail, we define R. Reiter's *regression operator* $\mathcal{R}$ [62] below to illustrate the underlying method used to perform goal-regression. However, we use the simplified version which accounts for regressable formulas that only mention relational fluents and no functional fluents. Note that a formula $W$ is regressable if and only if:

- Each *situation* term mentioned by $W$ has the form $do([A_1...A_n], S_0)$ whereby $n \geq 0$, and $A_1...A_n$ are action terms $A(\overrightarrow{t})$.
- Each *poss atom* mentioned by $W$ has the form $poss(A(\overrightarrow{t}), S)$ for some $n$-ary action function $A$.
- $W$ does not quantify over situations.
- $W$ does not mention any ordering relation or any equality atom for *situation* terms.

By this definition, the number of actions involved in *situation* terms can clearly be observed since the terms are rooted in $S_0$. Furthermore, the action function symbol is conveniently identified as the first argument of a given *poss atom*. In turn, given a regressable *poss atom*, it is replaced by the right-hand-side (RHS) of the action's precondition, whereby regression is continued on the expression. Additionally, given a regressable relational *fluent atom* about *do(A,S)*, it is replaced by the logically equivalent expressions about state $S$ as given by the *SSA*. The operator regresses until it cannot make any further replacements. In other words, the regression procedure is concluded

once each of the ground situation terms within the original regressable formula have been shown to be rooted within the initial state, $S_0$. The regression operator $\mathcal{R}$ is formally defined as follows:

1) $\mathcal{R}$ could result in four possibilities given a regressable atom $W$ that does not mention any functional fluents, but only relational fluents:

- $W$ is a situation-independent atom. In other words, $W$ is an equality between terms of the same sort *object* or *action*.

  Then $\mathcal{R}[W] = W$.

- $W$ is a relational fluent atom of the form $F(\overrightarrow{t}, S_0)$.

  Then $\mathcal{R}[W] = W$

- $W$ is a regressable *poss atom* of the form $poss(A(\overrightarrow{t}), S)$ for terms $A(\overrightarrow{t})$ and $S$ of sort *action* and *situation*, respectively. Here, $A$ is an action function symbol. Then, in BAT there must be an action precondition axiom for $A$ of the form $\forall s \forall \overrightarrow{x}.poss(A(\overrightarrow{x}), s) \leftrightarrow \Pi_A(\overrightarrow{x}, s)$. It is assumed that all quantifiers (if any) of $\Pi_A(\overrightarrow{x}, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $poss(A(\overrightarrow{t}), S)$.

  Then $\mathcal{R}[W] = \mathcal{R}[\Pi_A(\overrightarrow{t}, S)]$.

  Thus, replace the atom $poss(A(\overrightarrow{t}), s)$ with the RHS of the action $A$'s precondition axiom and continue regression on the that expression.

- $W$ is a relational fluent atom of the form $F(\overrightarrow{t}, do(A, S))$. Let *F's SSA* in BAT be of the form $\forall a \forall s \forall \overrightarrow{x}.F(\overrightarrow{x}, do(a, s)) \leftrightarrow \Phi_F(\overrightarrow{x}, a, s)$. Assume that all quantifiers (if any) of $\Phi_F(\overrightarrow{x}, a, s)$ have had their quantified variables renamed to be distinct from the free variables (if any) of $F(\overrightarrow{t}, do(a, s))$.

  Then $\mathcal{R}[W] = \mathcal{R}[\Phi_F(\overrightarrow{t} A, S)]$.

  Thus, replace the atom $F(\overrightarrow{t}, do(A, S))$ by a suitable instance of the RHS of the equivalence in $F$'s SSA, and regress this formula.

2) Regression is defined inductively for non-atomic formulas $W$ as follows:

- $\mathcal{R}[\neg W] = \neg \mathcal{R}[W]$
- $\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$
- $\mathcal{R}[\exists v(W) = \exists v.\ \mathcal{R}[W]$

Besides the exclusion of planning tasks which involve the use of functional fluents, we place additional restrictions on the domains we consider in this thesis. Moreover, we also exclude domains that involve context-dependent effects as part of their action schemas. This restriction is imposed as domains which involve conditional effects increase the complexity of the regression. Furthermore, we disregard domains that include derived predicates. Since these predicates are not defined by SSAs, the regression operation cannot be applied. Hence, we aimed to apply regression efficiently on relatively unsophisticated domains prior to expanding our dataset.

## 2.5 Related Work

This section details related work that which motivated and guided the central contribution presented within this thesis. These algorithms were examined thoroughly as they demonstrated potential value for our work. In turn, an implementation was developed in Prolog for each of the following algorithms according to our understanding, to the best of our ability.

With regards to solving classical planning problems, regression-based approaches, as in *Non-Partial Order Planning* (Unpop) [50] and the *Greedy Regression Table System* (GRT) [60] were shown to produce informative heuristics over popular forward chaining methods when first introduced. Over time, these approaches have advanced to combine aspects of other strategies in order to better direct search. For instance, McDermott was the first to explicitly introduce regression through forward search. This was achieved with the use of regression-match graphs in the Unpop planner. Subsequently, additional regression-based algorithms have been developed to further improve the course in which search is conducted. Furthermore, these improvements demonstrate competitive performance on benchmarks involved in previous IPCs against other planners. For these reasons, we chose to further investigate the advantages of regression-based approaches and similar strategies.

### 2.5.1 Regression (R)

The *R* planner developed by Fangzhen Lin [40] is a regression-based planner. Specifically, the planner relies on a sub-goaling strategy in order to produce the sequence of actions capable of satisfying the goal state from the initial state. Given the ground goal state $G$, the actions $A$ which produce goal atoms $g \in G$ as a positive effect, are identified. These actions are relevant in that they contribute towards achieving $G$. If an action $a \in A$ is applicable, then it is executed to produce a new situation. Otherwise, for each action $a$, the preconditions of the action *Pre(a)* are considered as new sub-goals. This procedure continues recursively until all sub-goals are satisfied in the current situation. All actions are assumed to be context-free (i.e., STRIPS actions). Additionally, the sequence of visited sub-goals is maintained in order to avoid cycles whereby a cycle is evident when a sub-goal is required to achieve itself.

The planner uses two mutually recursive functions, *FIND-PLAN(G)* and *FIND-PLAN-SG(g)*, that operate using a global database and two global stacks, $\Gamma$ and $\Pi$. The global database is initialized to consist of atoms in the initial situation. $\Gamma$ is used to store the sequence of actions that the planner is currently committing to. $\Pi$ is used to store the sequence of simple goals (i.e., literals) that the planner is currently attempting to satisfy. Both $\Gamma$ and $\Pi$ are initially empty. *FIND-PLAN(G)* calls upon *FIND-PLAN-SG(g)* in order to find an action that satisfies a simple goal $g$. Given that the action may contain variable arguments, it must compute the common ground preconditions according to literals that exist within the current state description. Then, if an action is found to satisfy $g$ and is applicable in the current situation, then it is executed and the global database is updated. Otherwise, the action's preconditions are pushed onto $\Pi$ to be regressed. Subsequently, the next goal literal $g'$ from $\Pi$ is selected and regressed with regards to the new situation. While attempting to satisfy a goal literal, a dead-end may be encountered whereby no actions can be found to satisfy it. In this case, the actions in $\Gamma$ are backtracked in an attempt to pursue another sequence that may satisfy the literal.

This algorithm offers a slightly different approach than how regression is traditionally applied. The formal regression procedure is concluded when a given goal is found to be satisfied in the initial situation. However, *R* planner updates the current situation by the actions which satisfy simple goals throughout the regression. In turn, the planner has shown to produce significant

results. Regarding the benchmarks used at the AIPS-2000 competition, *R* planner outperformed 12 competing teams for the *blocksWorld* domain. The most difficult problem included instances involving 45 blocks which required more than 150 actions to be solved. *R* planner was the only planner to optimally solve all problems from *blocksWorld* within the prescribed time limit.

However, the planner did not consistently perform well across other domains. Mainly, the planner produced sub-optimal plans for problems involving many goal literals which do not interact with each other. For instance, *R* planner performed poorly on the *gripper* domain, whereby an agent is tasked with moving balls from a given room to an adjacent room. Additionally, the agent has two available arms that enable it to carry a maximum of two balls at a time. Optimally, the agent should move two balls (i.e., one in each arm) with each transition from the adjacent rooms. However, *R* planner relies on a sub-goaling procedure that attempts to satisfy each sub-goal individually. Therefore, *R* planner does not take advantage of both arms. Rather, the agent transfers the balls one at a time. Hence, the sub-goaling procedure does not account for goal interactions well.

For our purposes, *R* planner was studied given that it was a regression-based planner which performed significantly well against other competing planners. Although, *R* did not consistently perform as well for some domains, the strategy showed potential in that regression based planners are capable of solving problems efficiently and optimally.

### 2.5.2 Regression and Goal Ordering (RO)

For a given classical planning task, the goal state is comprised of individual goal propositions that must be satisfied together through a sequence of actions applied from an initial situation. However, throughout the search, an applicable action may delete goals that were achieved previously by another action in the history. This undesirable interaction between the effects of actions is known as *goal-clobbering*. A second issue involves the violation of preconditions of some action since achieving one goal could result in a dead-end state.

Figure 2.5 illustrates the well-known *Sussman Anomaly* example involving the *blocksWorld* domain. For instance, consider three blocks labelled *A*, *B*, and *C*, such that *A* and *B* are placed on the table and that block *C* is placed on top of *A*. The desired block configuration states that (1) *A* is on top of *B* and (2) *B* is on top of *C*. To solve (1), *C* must be moved to the table, whereby, *A* can

then be placed on top of $B$. However, the second goal cannot be achieved unless by undoing the first goal. Similarly, if the $B$ is placed on $C$ first, it must be removed in order to solve the first goal. This example in particular portrays the difficulty experienced in planning since achieving either sub-goal individually prevents the ability to achieve the other.

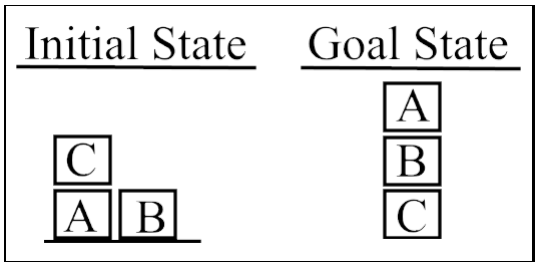



FIGURE 2.5: Illustrates the *Sussman Anomaly*

Furthermore, although goals don't always interact with each other, their ordering may be essential to solving the goal state as a whole. These issues are common across various domains, and in turn motivate research effort towards ordering goals throughout the planning procedure. These works explore common concerns, such as the method used to order two given goals [27, 29] and the method of extracting a total order on goal propositions in planning [31]. Another approach developed by Fangzhen Lin [39] demonstrates how information regarding subgoal ordering can be extracted from the background action theory. A prevalent approach to goal ordering involves the iterative achievement of sub-goals from a given set of ordered goals. For instance, given the ordered set $\{g_1,...,g_n\}$, $g_1$ may be achieved in a resulting state $s_1$ upon applying some action. Subsequently, an attempt is made to satisfy $\{g_1,g_2\}$ by proceeding from the state $s_1$. These procedure continues until a state $s_n$ is produced such that all sub-goals are satisfied in the state. These approaches vary in the method used to order the sub-goals and the strategy in which search is performed.

For our purposes of applying regression in planning tasks, we review the *Regression and Goal Ordering* (RO) planning algorithm developed by Razgon and Brafman [59]. RO applies a combination of regression and goal ordering techniques to direct each step in forward search. In turn, this approach reduces unnecessary and excessive backtracking by drawing greater attention to actions which are relevant to satisfying the goal. Typically, goal ordering is performed for the original goal only, however this work is among the first to apply a dynamic goal ordering approach.

RO performs standard forward search by applying actions starting from the initial state towards the goal state. However, the actions are selected according to the heuristic estimate computed by performing regression. Given the original goal $G$, the sub-goals are derived from the preconditions of actions that satisfy the goal. The sub-goals are then ordered according to the probability of being deleted by actions that achieve other sub-goals. Specifically, a sub-goal is assigned a higher priority based on the likelihood of it being destroyed when achieving another sub-goal. In turn, the sub-goal with highest priority is regressed using depth-first search. This recursive procedure continues until a given sub-goal is satisfied in the current state and an action can be executed to advance forward search. As a result, the sequence of actions which achieve the original goal is arranged more accurately through this regression and goal ordering procedure.

Similar to Lin's $R$ planner, regression is not concluded at the initial situation, but rather at the current situation which is updated by the final action identified through regression. However, RO differs in the way that the sub-goals are ordered and regressed. Specifically, given the list of sub-goals to be regressed, the sub-goal ordering procedure is as follows:

1) For each sub-goal, $q_i, ...q_n$, compute the probability $P(q_i)$ that $q_i$ will be destroyed while achieving all other sub-goals in the list.

2) For each probability $P(q_i)$,...,$P(q_n)$ that $q_i$ will be destroyed while achieving all other sub-goals:

   2.1) Attribute the highest priority to the sub-goal that is most likely to be destroyed when achieving other sub-goals. In other words, priority is given to the sub-goal associated with the highest value since its preferable to regress this sub-goal earliest, otherwise it will less likely be destroyed while regressing other sub-goals. In this way, the sub-goal that is most likely to be destroyed when achieving other sub-goals will be achieved latest in forward search. In turn, the last action involved in regression will guide the forward search more accurately.

This technique directs forward search to consider actions that are relevant at the current stage of search as they satisfy sub-goals sequentially leading back to the original goal. In addition, the action executed in forward search will likely achieve goals that won't be destroyed by subsequent actions.

RO was shown to be competitive with the popular and efficient *FF* planner when comparing results from the AIPS-2000 competition benchmark. Recall that *FF* is a forward search algorithm that uses GRAPHPLAN to solve the relaxed version of the problem. While RO outperforms *FF* for some domains, it takes more time and produces sub-optimal solutions for *blocksWorld*. The primary cause is due to the additional and redundant computational effort of repetitive states. RO does not guarantee an optimal plan but usually contains a near-optimal plan as a subsequence, whereby an optimal plan can be retrieved using a plan refinement algorithm.

In summary, RO combines forward search with backward regression and goal ordering. However, RO shares a similar issue as Lin's *R* planner and other regression-based planners in general. Specifically, when regressing on a sub-goal, all relevant actions that produce it as a positive effect are identified. In order to determine the action which satisfies the sub-goal greedily, we may need to consider a significant number of variations of action arguments for all actions. In other words, regression-based planners suffer from the similar limitations of performing forward search. However, as mentioned for RO, memoization can be used as an optimization technique to minimize the branching factor. By storing previously achieved information throughout the regression, the redundant computation of states can be avoided. In turn, time and memory resources can be better preserved.

### 2.5.3 Non-Partial Order Planner (Unpop)

D. McDermott introduced the use of regression-based heuristics in forward search with the development of *Unpop* (Non-Partial Order Planner) [50]. Briefly, Unpop performs forward search whereby, the distance from the current state to the goal is computed backwards by means of *regression-match graphs*. These graphs are constructed by regressing on the goals through actions that produce them as a positive effect. The process continues by regarding the preconditions of actions as new sub-goals. Subsequently, regression is applied on the actions' preconditions until all corresponding preconditions are satisfied by the current state. The heuristic is computed from the regression-match graph as the estimated minimum number of regression steps required to achieve the goals. Forward search is then directed by the heuristic and the procedure is repeated until the goal state is satisfied (if a solution exists).

Formally, Unpop operates in two phases: *forward search phase*, and *backward heuristic computation phase*. At each step of forward search, the frontier nodes $N$ are ranked according to the estimated effort required to satisfy the goal state from each $n \in N$, respectively. Accordingly, the regression phase is activated in order to generate the regression-match graph. The nodes within the graph correspond to goals and the actions which achieve them, respectively. The edges correspond to *regression* edges and *match* edges. The regression-match graph is used to assign the current state $n$ with the appropriate cost. Hence, given the goal state $G$, a goal literal $g \in G$ is selected without any particular ordering. Subsequently, the actions $A$ that produce $g$ as a positive effect are identified. These actions are easily identified given the PDDL specification for action schemas as shown in Figure 2.2. Furthermore these actions are connected to their respective goals using *regression* edges.

By definition, regression of a proposition $P$ through action $A$ is defined as follows [51]:

$$[A]^R(P) = \neg P \wedge [A]^{\overline{R}}(P) \vee P \wedge [A]^{\underline{R}}(P)$$

where $[A]^{\overline{R}}(P)$ is defined as the *causation precondition* for $P$ before executing $A$, and $[A]^{\underline{R}}(P)$ is the *preservation precondition* or $P$ before executing $A$. In other words, $[A]^{\overline{R}}(P)$ is defined as the weakest condition that causes $P$ to become true after applying $A$, given that it was false. The *causation precondition* is used to perform regression. Additionally, $[A]^{\underline{R}}(P)$ is defined as the weakest condition that maintains $P$ after applying $A$, given that it was true. The *preservation precondition* can be used to detect loops and terminate the regression for invalid paths.

To regress through the actions, regression is applied on the actions' preconditions. For instance, in *blocksWorld*, given a goal whereby *blockA* is to be made clear, this proposition can be regressed through the action which moves some block *?blockX* (that is on top of blockA) onto another block *?blockY*:

$$[(move\ ?blockX\ blockA\ ?blockY)]^{\overline{R}}\ (\ (clear\ blockA)\ )$$

The resulting formula to be regressed is the conjunction of the action preconditions:

$$(on\ ?blockX\ blockA) \wedge (clear\ ?blockX) \wedge (clear\ ?blockY)$$

However, Unpop does not necessarily ground the preconditions prior to applying regression. Specifically, three methods are considered when matching variables in the action preconditions to the objects of the given problem:

1) Variable arguments of the preconditions are maintained as global variables. The variables become instantiated throughout the remaining procedure. For instance, if an action in forward search causes the proposition *(on blockB blockA)* to become true, then the goal *(on ?blockX blockA)* can be satisfied later by grounding *?blockX* with *blockB* for all instances within the graph. However, the effort required to satisfy a precondition that includes variables is not known until it is grounded. Hence, it is advantageous to ground the variables as they are presented in order to compute an accurate estimate.

2) Variable arguments of the preconditions are instantiated using all *non-perverse* combinations of objects of the problem. Whereby, non-perverse preconditions maintain appropriate types of objects and auxiliary conditions. An example of a perverse precondition is On(Table, Table). However, this method may produce a significant number of irrelevant preconditions.

3) Variable arguments of the preconditions are instantiated with objects given the literals in the current state. For this method, the algorithm attempts to ground variables by maximally matching the preconditions with the current state. This method reasonably estimates the effort required to satisfy the goals. Regression is applied through all actions that produce a goal in order to identify the path which satisfies the goal greedily (i.e., with the least number of actions).

The resulting formula consists of a conjunction of (partially ground) preconditions, that are to be regressed relative to the current state. Furthermore these preconditions are connected to their respective actions within the graph using *match* edges.

With regards to the computation of the estimated effort *EE* required to satisfy goals, it is recursively defined as follows:

- If $p$ is a goal literal, that is true in the current state, then

  $EE(p) = 0$

- If $p$ is a goal literal, that is **not** true in the current state, then

  $EE(p) = 1 + min_{g \in reductions(p)} EE(g)$

  Whereby, a *reduction* is a set of goal conjunctions (i.e., preconditions) obtained by regressing through an action that causes $p$ to become true.

- If $g$ is a a goal conjunction, then

  $EE(g) = min_{h \in cohorts(g)} \Sigma_{p \in h} EE(p)$

  Whereby a *cohort* represents a set of preconditions belonging to the same action.

- Otherwise, the effort is $\infty$.

Hence, if a goal literal $p$ is satisfied in the current state, then the effort required to achieve $p$ is zero. The algorithm then continues to attempt to achieve the remaining goals. Otherwise, Unpop applies regression to satisfy all preconditions of actions $A$ that achieve $p$. In other words, Unpop recursively attempts to satisfy each precondition of all actions as a new sub-goal. If successful, then the estimated effort of achieving a goal literal $p$ is calculated as the minimum estimated effort required to achieve the preconditions of an action $a \in A$ plus the action cost. Furthermore, the estimated effort of achieving the preconditions of an action (i.e., goal conjunction) is the minimum sum of efforts for each precondition.

The estimated effort is assigned to the respective frontier search node, whereby Unpop proceeds with the forward search. The planner pursues the path of the action with the least estimated effort cost until a plan is found to satisfy the goal state from initial state.

To illustrate the regression-match graph, we produce an example from the *blocksWorld* domain. Note that it is possible to express the same domain in multiple ways. Normally, a description of *blocksWorld* includes the fluent *HandEmpty(s)* to represent that the arm is not holding a block. However, the following example characterizes the domain with a minimal description of actions

and fluents. This simplification is performed for convenience whereby blocks can be moved immediately to and from the table, or onto other blocks without the need of an arm. For this representation, we define three actions as shown:

**Actions:**

**i)** *move(BlockA, BlockB, BlockC)*: move *BlockA* from *BlockB* onto *BlockC* iff *BlockA* is on *BlockB*, *BlockA* is clear, and *BlockC* is clear (i.e., *On(BlockA, BlockB, s)*, *Clear(BlockA, s)*, *Clear(BlockC, s)*).

**ii)** *moveFT(BlockA, Table, BlockB)*: move *BlockA* from the table onto *BlockB* iff *BlockA* is on the table, *BlockA* is clear, and *BlockB* is clear (i.e., *On(BlockA, Table, s)*, *Clear(BlockA, s)*, *Clear(BlockB, s)*).

**iii)** *moveTT(BlockA, BlockB, Table)*: move *BlockA* from *BlockB* onto the table iff *BlockA* is on *BlockB*, and *BlockA* is clear (i.e., *On(BlockA, BlockB, s)*, *Clear(BlockA, s)*).

We define two fluents as shown:

**Fluents:**

**i)** *Clear(BlockA, s)*: *BlockA* is made clear iff another block *BlockB* on top of *BlockA* is moved to the table or moved to another block *BlockC*. Otherwise *BlockA* remains clear unless another block *BlockB* is moved on top of *BlockA*.

**ii)** *On(BlockA, Y, s)*: *BlockA* is on *Y* (where *Y* can be another block *BlockB* or the table) iff *BlockA* is moved onto *Y*. Otherwise, *BlockA* remains on *Y* unless it is moved from *Y*.

For this example, the initial state and goal state are displayed in the following Figure 2.6 whereby the initial state is { *Clear(C, $S_0$)*, *On(C, B, $S_0$)*, *On(B, A, $S_0$)*, *On(A, Table, $S_0$)* } and the goal state is { *Clear(A, s)*, *On(A, B, s)*, *On(B, C, s)*, *On(C, Table, s)* }.

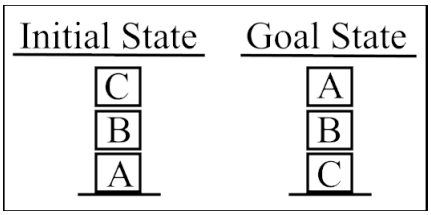



FIGURE 2.6: Example from the *blocksWorld* domain

The regression-match graph illustrated in Figure 2.7 is based on the *blocksWorld* example shown in Figure 2.6. The graph is produced with respect to the initial state. This example grounds variables in the action preconditions according to the third approach as described above. Whereby,

variables are instantiated with objects to maximally match the preconditions with the current state. Also, this example imposes a bound of 3 steps for regression.

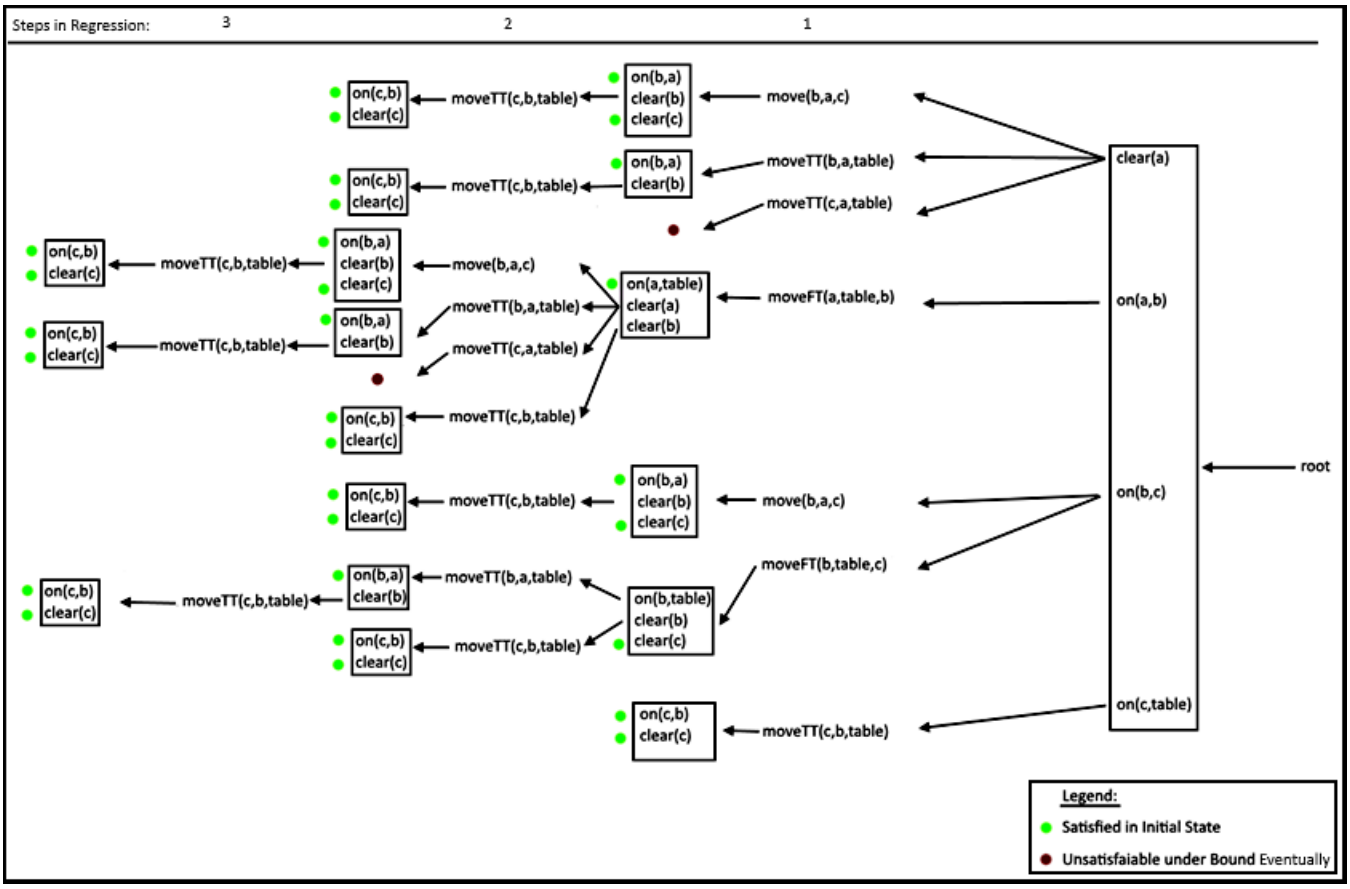



FIGURE 2.7: Illustrates the *regression-match graph* for the *blocksWorld* example in Figure 2.6

Forward search is performed by two methods: best-first, and hill-climbing search with restarts. Best-first search is applied by maintaining nodes in the search frontier as a priority queue. The queue consists of plan prefixes starting from the initial state. The node evaluated with the lowest estimate is given highest priority. This node is expanded, whereby its neighbours are evaluated and inserting in to the queue accordingly. However it is shown that the heuristic estimate is not significantly informative at early stages during the search. This is due to the inaccuracies within the regression-graph given that the graph can become significantly large. Furthermore, neighbouring nodes will result in similar estimates. As a result, best-first search will behave comparably to breadth-first search early during search. However, this inefficiency can be avoided by

employing hill-climbing search. Whereby, the most promising node in the frontier is pursued. The search restarts given the current situation can no longer be expanded. The node chosen for continuing the search is chosen randomly from a local state that was given highest priority at some previous state. Another promising approach involves applying forms of exploration as in [41] when the heuristic estimates across states in the frontier are not significantly informative. In this work, Geffner and Lipovetzky develop a *best-first width search* (BFWS) scheme that applies directed search with respect to novel atoms produced, rather than search that is oriented by the goal. This work has been shown to produce results which outperform state-of-the-art planners.

At the time of its development, Unpop exhibited significant performance when compared to other planners. The regression is performed without taking goal interaction into account. In turn, the heuristic is computed inexpensively in general. Additionally, goal conjunctions are constrained in such a way to entail as many individual goals as possible. The results were produced by a hybrid of both best-first and hill-climbing search techniques. Unpop generally took more time to solve problems at the AIPS-98 comparatively. Although it is important to note that it was written in Lisp, whereas competing planners were written as C programs. However, Unpop was able to find more solutions to exceedingly large problems compared to other planners. This feature was mainly due to the regression technique, as forward search planners significantly consider more actions. Similar to Lin's *R* planner, Unpop relies on regression. However Unpop applies regression in order to provide a heuristic, as opposed to directly producing a plan. Furthermore, Unpop relies on regression-match graphs to (partially) instantiate preconditions of actions involved in regression. In contrast, *R* planner simply performs backwards search by applying regression combined with an effective sub-goaling strategy.

### 2.5.4 The Greedy Regression Table (GRT)

Given that the *Greedy Regression Table System* (GRT) plays a significant role in our work, a thorough explanation of the system is provided. The GRT developed by Refanidis and Vlahavas [60, 61] is a forward search, backward heuristic construction planner. GRT operates in two phases. First, the *pre-processing phase* is responsible for generating the heuristic information by applying backward search. The heuristic is computed once and the information is stored into a table. Subsequently, the *search phase* proceeds in the forward direction by consulting the table at each step

on a best-first basis. GRT computes the heuristic only once as opposed to other planners that compute the heuristic in the forward direction. Additionally, performing forward search prevents the planner from encountering invalid states.

**Pre-processing Phase**

Briefly, the *pre-processing phase* applies *inverted* actions to advance the search towards the initial state from the given goal state. When an inverted action is applied, its positive effects are defined as *facts* and inserted into the table. Each fact is assigned a distance to the goal state based on the number of actions applied previously. Also each fact is assigned with a set of *related facts*. This phase terminates once the atoms that are true in the current state are satisfied by the initial state. In greater detail, the *pre-processing phase* operates as follows:

**0.1) Generate the set of inverted actions**: In order to perform backward search, actions $A$ given in the domain must be inverted into the set $A'$, such that, for each action inverted $a' \in A'$:

**i) Pre(a') = Add(a) $\cup$ Pre(a) \Del(a)**: the precondition of $a'$ is the union of the original action's positive effects and preconditions which are not deleted (i.e., non-delete preconditions). If an action $a$ is to be applied backwards, then $a'$ must require the positive effects produced by $a$. Additionally, $a'$ must require literals that were conditions of $a$ which remained true after its execution.

**ii) Del(a') = Add(a)**: the negative effects of $a'$ are defined as the positive effects of the original action. This is to restore the state prior to the addition of literals (by the original action) by removing literals that were added.

**iii) Add(a') = Del(a)**: the positive effects of $a'$ are defined as the negative effects of the original action. This is to restore the state prior to the deletion of literals (by the original action) by adding literals that were removed.

**0.2) Initialize the GRT**: The GRT is initialized to contain facts about the goal literals. For each goal literal in the goal state, a fact is defined to contain the literal, a distance of zero to the goal state, and an empty set of related facts. The facts are inserted into the GRT.

**0.3) Enhance the goal state**: The goal state may be incomplete since it is represented by partial-truth assignments. Meaning that, the goal state does not always explicitly make a reference to all objects in the initial state of the planning task. In order to perform backwards search, the

goal state must be enhanced to involve all objects in the problem so that they may be used while performing backward search. The techniques used to enhance the goal state are explained in the next subsection.

**0.4) Perform backward search**: Starting from the enriched goal state, apply inverted actions that are executable in the current state towards the initial state. Upon executing an action, populate the greedy regression table with the respective facts. This phase terminates once all atoms that exist in the current state are satisfied by the initial state, upon which, the GRT is returned.

**0.4.1) Populate the GRT:** Upon applying an inverted action $a'$, identify its respective positive effects. Define a fact $\mathcal{F}$ for each positive effect $p \in Add(a')$ of $a'$. Next, each $\mathcal{F}$ is assigned its distance to the goal state (i.e., number of actions previously applied). Furthermore, $\mathcal{F}$ is assigned the set of its related facts $rel(\mathcal{F})$. The related facts are explained more thoroughly further below. At the moment, consider the related facts $rel(\mathcal{F})$ as facts that were previously achieved in order to achieve the given fact $\mathcal{F}$.

**0.5) Apply best-first search**: forward search is performed by consulting the GRT produced in the *pre-processing phase* to better direct the search. The heuristic computation is described more comprehensively further below.

#### Goal State Enhancement

In the case that a given goal state does not represent a complete state by explicitly referencing all objects of the planning task, it must be enhanced. This enhancement is performed so that inverted actions can make use of available objects while performing backward search. Note, this enriched goal is no longer used once the the backward search is complete. For clarification, the original goal state is used when performing forward search during the *search phase*.

In general, a goal state is enhanced by including new literals that do not contradict literals within the goal state. The goal enrichment procedure will generate new literals by performing one of three methods shown below:

**1)** Produce new literals for all objects in the initial state that do not contradict goal literals. Consider all new literals as goal facts with zero distances and empty sets of related facts within the GRT. This approach allows for a large number of actions to be applicable which in turn allows the production of the heuristic to be faster. However, the heuristic is less useful as estimates

become similar to each other. Also, the additional effort spent in producing these new literals will impose an overhead that decreases efficiency.

**2)** Only literals included within the initial state are used to enhance the goal state. In turn, this method will produce a more informative heuristic since the distances between facts will be greater than that of the first method. However, this approach is not reliable in general as the initial state may only consist of literals which contradict literals in the goal state.

**3)** New literals are added dynamically in the case that no inverted action can be applied. This method results in better plans than the previous methods as it will not saturate the GRT with facts that do not assist the *pre-processing* phase. Nevertheless, the search is not likely to produce optimal or near-optimal plans in general given the performance is dependant on the given problem.

#### Related Facts

When computing the heuristic during the *search phase*, GRT accounts for interactions between facts since achieving a fact possibly affects the achievement of other facts. Initially, it is stated that a fact $q$ is related to another fact $p$, if achieving $p$ causes fact $q$ to be achieved as well. This notion is extended with the following definition: the set of all related facts to a specific fact $p$ given an inverted action $a'$ achieving fact $p$ is:

$rel(p) = Pre(a') \cup rel(Pre(a')) \cup Add(a') \setminus Del(a')$

The related facts play an essential role in accurately estimating the distance of a given state to the goal state during forward search. Briefly, the GRT is consulted using the literals that are true in the given state. The distance is calculated by cross-referencing the literals against the facts within the table. In order to adequately express the idea behind the use of related facts, we first illustrate the method used to estimate the distance between a given state and the goal state by consulting the GRT. The procedure used to compute the heuristic estimate is called the *Aggregate Algorithm*.

During the *search phase*, the search frontier is expanded in the forward direction from initial state to the goal state. In order to evaluate the distance from a given state to the goal, the literals composing the state are obtained. The literals are then cross-referenced against the facts in the GRT. If a literal exists as a fact, then it is placed into a list. Given the list of facts, the distance of the current state to the goal state is computed as follows:

**Case 01**: For each fact $f_i$ in the list of facts, if $f_i$ exists as a related fact of another fact $f_j$, and $f_j$ exists as a related fact of $f_i$, then they are grouped together so that their distance is only considered once. In other words, if two facts are related to one another, then they must have been achieved together as positive effects of an inverted action. In turn, their distances should only be factored in once. Hence, the facts are grouped together.

**Case 02**: For each fact $f_i$ in the list of facts, if $f_i$ exists as a related fact of another fact $f_j$, and $f_j$ does not exist as a related fact of $f_i$, then the distance of $f_i$ is not considered. The distance of $f_i$ is ignored since it was achieved when achieving $f_j$, thus its distance is reflected in the distance of $f_j$. In other words, if a fact is related to some other fact that is not reciprocally related in kind, then the fact's distance should not be considered since it was achieved previously by the other fact.

**Case 03**: For each fact $f_i$ in the list of facts, if $f_i$ does not exist as a related fact of another fact, then it is grouped individually and the distance of $f_i$ is considered since it was not achieved when achieving another fact.

The final estimate is computed as the maximum distance within each grouping of facts. Thus, upon demonstrating the application of related facts, we now re-examine its definition. Given a fact $p$ achieved by an inverted action $a'$, the set of related facts of $p$ is the union of:

**1) Pre(a')**: the facts required to apply $a'$. Hence, when calculating the heuristic estimate of the current state to the goal state, the distance of a literal will not be included in the estimate if it exists as a precondition of an inverted action required to achieve another literal in the current state. The distance is ignored since it will be reflected in the other literal that contains it within its set of related facts.

**2) rel(Pre(a'))**: related facts of facts required to apply action $a'$. Given a set of facts, $P$, the set of $P$'s related facts is the union of the related facts of $P$:

$$rel(P) = \cup_{p \in P} rel(p)$$

Similarly to the previous case, the distance of a literal will not be included in the estimate if it exists as a precondition of an inverted action required to achieve another literal. This prevents the redundant addition of costs since the literal's distance will be reflected in the fact that contains it within its set of related facts.

**3) Add(a')**: facts that have been achieved along with $p$. Facts achieved together as positive effects of the same inverted action are related to ensure that the cost of achieving these facts is considered once as they were achieved together.

**4) \Del(a')**: not including facts deleted by an inverted action $a'$. Negative effects of an inverted action are not related to $p$ as they do not remain with $p$ upon applying $a'$. In other words, not all facts that are deleted by an inverted action are relevant. Recall, that the negative effects of an inverted action are the positive effects of the original action. Hence, it is accurate to assume that not all positive effects of an original action are relevant in producing the goal state. For this reason, they are omitted from the set of related facts.

To summarize, the interaction of goals is managed through the notion of related facts. As a result, the heuristic estimate of a given state during forward search is more accurately representative of the true distance to the goal state.

**Example: Backward Heuristic Construction**

To illustrate the construction of the GRT, we demonstrate the backward search procedure using the same example shown in Figure 2.6. Since the goal state is complete, it does not require any enhancement. Next, the actions are inverted as specified by the definition given previously. Note that any mention of an inverted action by name will be preceded by the symbol $\mathcal{I}$ to avoid confusion. For example, observe that $\mathcal{I}$*moveFT* will in fact move a block **to** the table. Also, $\mathcal{I}$*moveTT* will move a block **from** the table. Lastly, the inverted action for $\mathcal{I}$*move* will perform the reverse of action *move* since its positive and negative effects have been exchanged.

The inverted actions are applied backwards from the goal state to the initial state. The optimal solution to this task requires 3 steps whereby the solution produced in backward search using inverted actions are: [$\mathcal{I}$*moveFT(a, table, b)*, $\mathcal{I}$*move(b, a, c)*, $\mathcal{I}$*moveTT(c, b, table)*]. Table 2.1 illustrates the GRT produced using this sequence of inverted actions. Note that for the remainder of this example, the situation argument within the fluents is omitted for readability. The resulting table stores facts that include literals which are relevant to producing the goal state from the initial state. Recall that the literal in each fact is derived from the positive effects of inverted actions. Associated with each fact is its respective distance to the goal state (calculated as the number of inverted actions applied thus far) as well as its set of related facts.

| Facts | Distance to Goal | Related Facts |
|---|---|---|
| clear(a) | 0 | [] |
| on(a,b) | 0 | [] |
| on(b,c) | 0 | [] |
| on(c,table) | 0 | [] |
| on(a,table) | 1 | [clear(a), clear(b)] |
| clear(b) | 1 | [clear(a), on(a,table)] |
| on(b,a) | 2 | [clear(b), on(a,table), clear(c)] |
| clear(c) | 2 | [clear(b), on(a,table), on(b,a)] |
| on(c,b) | 3 | [clear(c), on(a,table),on(b,a),clear(a)] |

TABLE 2.1: Illustrates the GRT produced for the *blocksWorld* example in Figure 2.6

The GRT in this example is constructed as follows:

**0)** The GRT is initialized with facts consisting of the goal literals of the problem such that their respective distances to the goal are zero, and their respective sets of related facts are empty.

**1)** Given the goal state, the only applicable inverted action is $\mathcal{I}$*moveFT(a, table, b)* whereby block *a* is moved to the table, and block *b* is made clear, as positive effects. A fact is defined for each positive effect and inserted into the table. Each fact is assigned a distance of one step to the goal. For each fact, its set of related facts include the preconditions of the inverted action that achieved it = {*on(a,b), clear(a)*}, the related facts of the preconditions = {}, other literals achieved together as positive effects = {*on(a, table), clear(b*)} excluding the negative effects = {*on(a, b)*}.

**2)** Consider the next inverted action applied $\mathcal{I}$*move(b, a, c)* whereby block *b* is moved from block *c* onto block *a* and block *c* becomes clear. As previously, a fact is defined for each positive effect and inserted into the table. Each fact is assigned a distance of two steps form the goal since this is the second action to be applied thus far. Again, the related facts for each fact consist of the preconditions of the inverted action that achieved it = {*clear(a), on(b,c), clear(b)*}, the related facts of the preconditions = {*on(a, table), clear(a*)}, other literals achieved together as positive effects = {*on(b,a), clear(c)*}, excluding the negative effects = {*clear(a), on(b,c)*}.

**3)** Consider the next inverted action applied $\mathcal{I}$*moveTT(c, b, table)* whereby block *c* is moved from the table onto block *b*. As previously, a fact is defined for the positive effect and inserted into the table. The fact is assigned a distance of three steps form the goal since this is the third action to be applied thus far. Again, the related facts for each fact consist of the preconditions of the inverted action that achieved it = {*on(c, table), clear(b), clear(c)*}, the related facts of the

preconditions = {*on(a, table), clear(a), clear(b), on(b,a)*}, other literals achieved together as positive effects = {on(c, b)}, excluding the negative effects = {on(c,table), clear(b)}.

Throughout backward search, many actions may achieve the same fact. However, maintaining duplicate facts and their respective related facts would negatively affect efficiency as well as the accuracy of the heuristic. Hence, only the fact which corresponds to the shortest (most greedy) path from the goal state is maintained. Preserving the fact with the least distance will in turn guide the search more accurately during forward search.

**Example: Forward Search Phase**

The *search phase* is performed starting from the initial state towards the goal state using best-first search. For each step in the expansion of the frontier, applicable actions are prioritized according to the estimated distance of the resulting states to the goal state. This estimate is computed by consulting with the GRT using the *Aggregate Algorithm* [61]. An example of prioritizing actions in forward search using the GRT shown in Table 2.1 is as follows:

**0)** The distance of the initial state to the goal is computed by consulting the GRT and using the *Aggregate Algorithm*. The computation for this state in particular is trivial; rather the computation for the next step is more appealing and described in more detail. The resulting distance of the initial state to the goal state is 3 which is accurate as three steps are required to satisfy the goal.

**1)** From the initial state, the only applicable action is *moveTT(c,b,table)*. Upon execution, the resulting state consists of five literals: {*clear(c), on(c,table), clear(b), on(b,a), on(a,table)*}. By consulting the GRT, the respective fact for each literal is identified within the table. The *Aggregate Algorithm* is applied:

**i)** *clear(c)* is grouped with *on(b,a)* as these facts are included in each other's set of related facts. As a result, the distance of this group is 2.

**ii)** *on(c,table)* has a distance of zero.

**iii)** *clear(b)* exists in the set of related facts of other facts (i.e., *clear(c)* and *on(b,a)*) whereby they are not included in the set of related facts of *clear(b)*. As a result the distance of *clear(b)* is not included since its distance is reflected in the distance of other facts.

**iv)** *on(a,table)* exists in the set of related facts of other facts (i.e., *clear(c), clear(b), and on(b,a)*) whereby they do not exists in the set of related facts of *on(a,table)*. As a result the distance of *on(a,table)* is not included since its distance is reflected in the distance of the other facts.

Thus the resulting distance of the current state to the goal state is 2 which represents the accurate (minimum) number of steps remaining to achieve the goal state. The search is continued until the goal state is satisfied by the current state.

**GRT Growth and Complexity**

Upon successful termination of the *pre-processing* phase, the number of entries that the GRT will possess is at least as many as the number of literals that exist within the initial state since the algorithm terminates once the current state is satisfied within the initial state in backward search. Hence, the GRT could potentially grow beyond the number of literals that exist within the initial state by applying unnecessary intermediate actions in an attempt to reach the initial state. Although, there is effort spent in moderating this growth by identifying the solution greedily. Specifically, in the case that an action adds a fact which already exists within the GRT, the algorithm will abandon the path by said actions as it results in a less optimal path.

The size of the related facts for a given fact $p$ is limited to the number of facts achieved before $p$ within the GRT. For an inverted action $a'$ that achieves fact $p$, the size of its related facts primarily grows due to the related facts of the preconditions required to apply $a'$ (i.e., *rel(Pre(a'))*). Furthermore, since all *rel(Pre(a'))* must have been produced prior to achieving $p$, there is no need to recalculate them. Instead, only a table look-up is required to identify the related facts.

GRT demonstrates competitive performance against other planners on several domains. The results show that no planner dominates in performance over all other planners. However, the results produced by GRT were consistently promising for most domains. For instance, GRT produced a solution in the fastest time for the *gripper* domain. GRT was able to produce more optimal plans than other planners (except FF) for the *blocksWorld* domain. Additionally, larger problems consisting of up to 33 blocks were solved whereas other planners could not solve problems with more than 20 blocks. Generally, GRT and FF were among the fastest planners in comparison. Furthermore, the quality of plans produced by GRT were better than other planners across several domains. The quality in performance was mainly attributed to the fact that GRT only requires

to compute the heuristic once as opposed to other approaches such as FF. In turn, this approach demonstrates great potential for our purposes by providing a relevant and accurate heuristic while preserving valuable computational effort. Furthermore, the GRT exhibits a considerable inefficiency when constructing the table. Specifically, the table is constructed as a result of performing uninformed backwards search. This approach does not make use of directed search, rather it applies uninformed search until the initial state is satisfied. Whereas, the actions required to satisfy the goal can be produced directly through goal-regression. As a result, the GRT can be constructed in a more efficient manner by taking advantage of regression.

### 2.5.5 GRAPHPLAN (GP)

GRAPHPLAN, developed by Blum and Furst [2, 3], performs forward heuristic best-first search. Briefly, the graph-based reachability heuristic is calculated by alternating between two phases: *graph expansion phase*, and *solution extraction phase*. The expansion phase generates and extends the *planning graph* from the current state. Whereby, all applicable actions in the current state are executed, while maintaining only their positive effects and all propositions achieved previously. The graph consists of alternating proposition and action layers from the current state to the goal state. The expansion proceeds until the goal state is satisfied or two consecutive identical proposition layers have been produced, in which case no solution exists. Upon successful termination, the proposition layers will consist of the conditions required to determine a relaxed plan. Upon generating the graph, the solution extraction phase determines a solution to the relaxed problem by performing backward search on the planning graph. The solution cost serves as an admissible heuristic estimate for forward search. Algorithm 2.1 briefly illustrates the GRAPHPLAN procedure.

In further detail, the planning graphs begin with the propositions that are true in the initial state. These literals compose the nodes in the $zero^{th}$ proposition level. The next layer consists of nodes that represent applicable actions given that their preconditions are satisfied in the preceding proposition layer. Each layer of nodes is connected by directed edges. Proposition nodes connect to subsequent action nodes given that they are preconditions of the action. Likewise, action nodes connect to subsequent proposition nodes that belong to the actions' positive effects, respectively.

Propositions achieved previously are maintained in each subsequent proposition layer through persistence actions in the intermediate action layer.

---

**Algorithm 2.1:** GRAPHPLAN

---

**Input:** the initial state $S_0$ and goal state $G$.

**Output:** the planning graph.

1. **GRAPHPLAN:**
2. *PlanGraph* = $p \in S_0$ % initialize planning graph to all initial state propositions
3. **foreach** goal proposition in the final layer
4. **if** all proposition are non-mutex (i.e., can be achieved together) **then**
5. *RelaxedPlan* = solution_extraction(PlanGraph)
6. **return** *Estimate* = length(*RelaxedPlan*)
7. **else if** all propositions are identical to previous layer **then**
8. **return** failure
9. **else**
10. *PlanGraph* = graph_expansion(*PlanGraph*)
11. proceed to *step 3*

---

Figure 2.8 displays the resulting planning graph produced for the *blocksWorld* example shown in Figure 2.6. The figure illustrates the alternating proposition and action layers of planning graphs. Actions are highlighted in blue. They connect to their preconditions with preceding lines to the respective ground literals in the previous proposition layer. They also link to the literals they achieve as positive effects in the successive proposition layer. All propositions are highlighted in orange aside from those within the final layer. Specifically, goal propositions are highlighted in green within the final layer. Mutually exclusive, *mutex*, actions and propositions are connected by yellow lines (not all *mutex* relations are shown for readability).

Actions are not applicable together if they are mutex, whereby *mutex* is defined as the mutual exclusion of binary relations between nodes of the same layer. Meaning that, an action $a_1$ cannot be

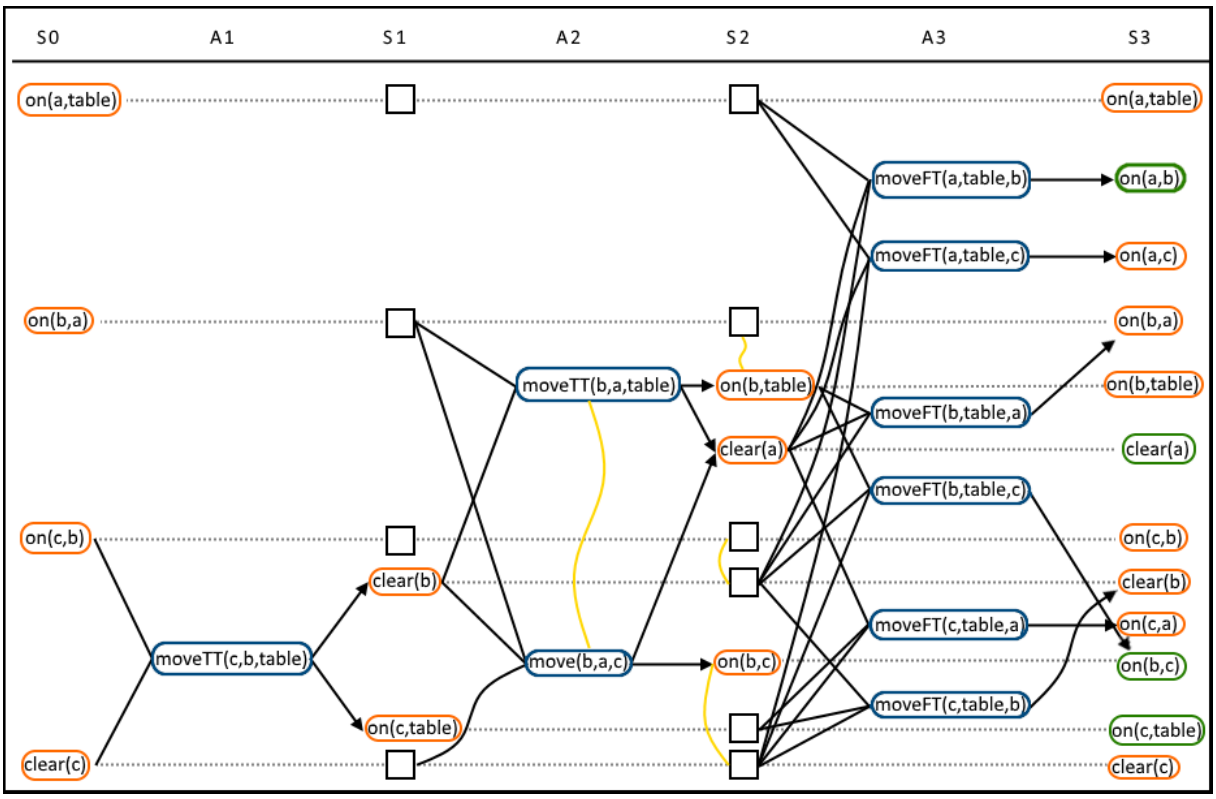



FIGURE 2.8: Illustrates the planning graph for the *blocksWorld* example in Figure 2.6

applied if it deletes a precondition or effect of another action $a_2$. Additionally, $a_1$ and $a_2$ are mutex if their preconditions in the preceding proposition layer are mutually exclusive. Specifically, a pair of propositions is mutex if all ways that actions (in the preceding layer) can achieve them are pairwise mutex. This concept is used when extracting the relaxed plan to ensure that actions remain consistent. The solution extraction phase is performed once the planning graph satisfies all goal propositions and that they are mutually consistent (non mutex) in the final layer.

Subsequently, the goal state is decomposed in which a backward extraction algorithm determines the set of consistent actions at each layer which contributes to the goal. The solution extraction phase operates as follows. For each goal $g \in G$ in the final proposition layer, an action $a$ that achieves $g$ is selected from the preceding layer. In turn, additional actions chosen at this level for subsequent goals are maintained if they are not mutually exclusive with other actions maintained thus far. Otherwise, the search backtracks to reselect another action from a previous iteration. Once a set of consistent actions are found for this level, the search recursively attempts to determine the solution for the actions' preconditions, respectively. The procedure successfully

terminates for a given action if its preconditions are satisfied in the initial state. If a solution cannot be found, then the planning graph is extended by an additional action and propositional level. The extraction phase then performs another attempt at extracting the relaxed plan. Algorithm 2.2 briefly illustrates the solution extraction procedure [5].

---

**Algorithm 2.2:** solution_extraction

---

**Input:** the planning graph $PG$, the goal state $G$.

**Output:** the relaxed plan.

1. **solution_extraction:**
2. $n$ = index of last layer in $PG$, such that $P_n$ is the last proposition layer
3. **foreach** $g \in G \cap P_n$
4. $P_n^{RP} = P_n^{RP} \cup p$
5. **foreach** i = {$n$...1}
7. **foreach** $p \in P_i^{RP}$
8. produce all actions $a \in A_{i-1}$ such that $p \in Add(a)$
9. $A_{i-1}^{RP} = A_{i-1}^{RP} \cup a$ % while assessing for mutual exclusivity
10. **foreach** $a \in A_{i-1}^{RP}, p \in Pre(a)$
11. $P_{i-1}^{RP} = P_{i-1}^{RP} \cup p$ % while assessing for mutual exclusivity
12. *RelaxedPlan* = $[P_0^{RP}, A_0^{RP}, P_1^{RP}, ..., A_{n-1}^{RP}, P_n^{RP}]$
13. **return** *RelaxedPlan*

---

By following this procedure, the relaxed plan is extracted, whereby the heuristic estimate is then computed as the length of the plan. Figure 2.9 visually represents the relaxed plan as a sub-graph of the planning graph illustrated in Figure 2.8. The relaxed plan consists of the following actions: [ *moveTT(c,b,table)*, *move(b,a,c)*, *moveFT(a,table,b)* ], whereby the heuristic estimate is 3 steps, which accurately represents the true distance to the goal state.

However, the planning graphs must be generated at each step in forward search since planning is performed from the complete current state to the partial goal state. Meaning that there

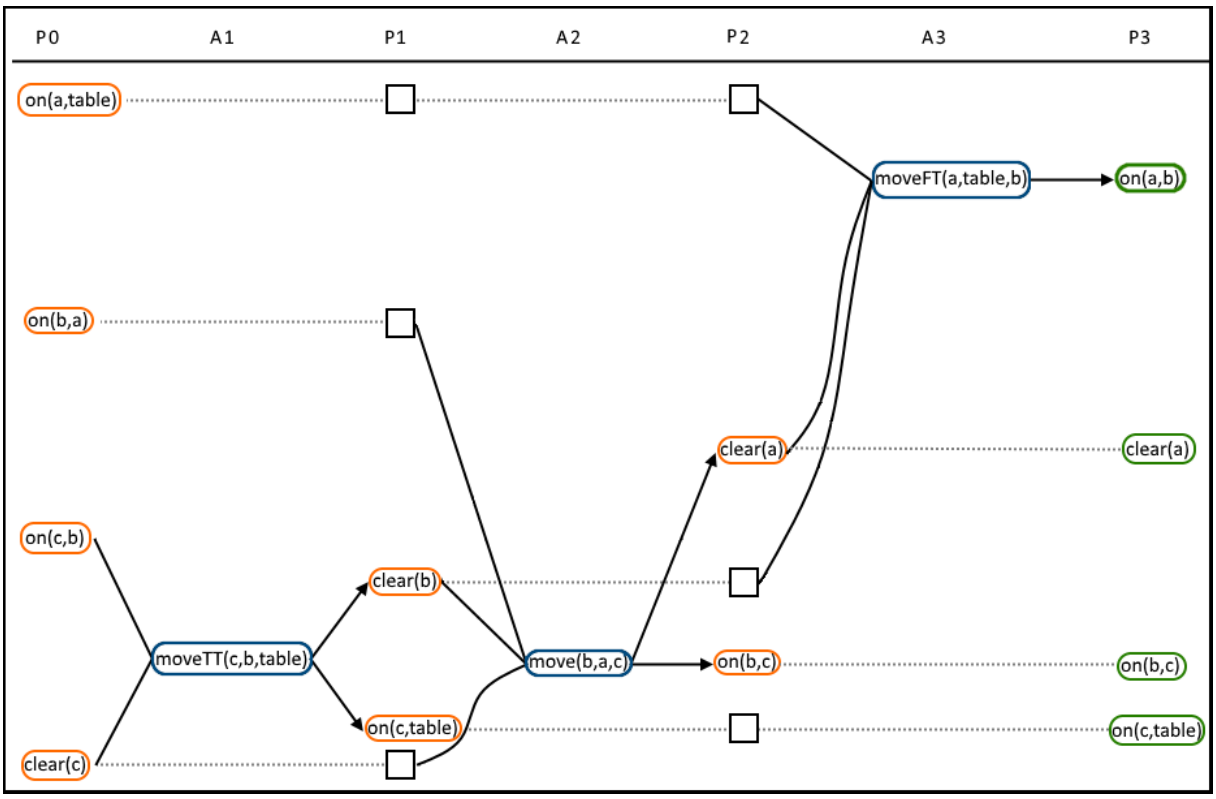



FIGURE 2.9: Illustrates the relaxed plan sub-graph of the planning graph shown in Figure 2.8

can be several goal states which all include the goal propositions in addition to other propositions produced in forward search. Hence, a considerable amount of computational effort is spent in producing the heuristic for each state, especially for a combinatorial state-space. However, the relaxed plan is relatively inexpensive to extract [5]. Furthermore, the estimate is shown to be more informative than the *additive*, *max*, and *set-level* heuristics (i.e., the index of the earliest proposition layer which includes all goals). The heuristic is admissible, thus guaranteeing optimality of a solution. As a result, GRAPHPLAN's development of mutual exclusion relations has consistently been applied by state-of-the-art planners. The algorithm significantly prunes a considerable portion of the state-space. Due to its performance, GRAPHPLAN was established to be among the most ideal methods of computing informative heuristics.

# Chapter 3

# Organic Chemistry Synthesis

This chapter includes a brief background of organic chemistry and organic chemistry synthesis and the motivation behind working with this domain as a planning problem. The methods used to manage the challenges posed by this domain are described. We present the organic chemistry synthesis benchmark based on organic chemistry examination questions from the Massachusetts Institute of Technology. Furthermore, we present the experimental results produced by applying two depth-first iteratively deepening search (DFIDS) solvers on the benchmark: i) standard DFIDS solver, and ii) BinTree solver. These solvers differ by means of representing and maintaining the state throughout the planning procedure. We then compare and analyse the results produced by both solvers. Lastly, we discuss an attempt to improve the results by employing heuristics within the BinTree solver in order to better guide the search.

## 3.1 Background

### 3.1.1 Organic Chemistry

Briefly, we review the fundamentals of organic chemistry and organic chemistry synthesis for our purposes. Formally, organic chemistry is the subdiscipline of chemistry which studies organic compounds containing carbon atoms. Organic compounds are key for all living things given that carbon atoms are the essential component. Atoms, such as carbon, are defined to be the fundamental units that play a role in forming molecular compounds. Atoms are composed of nuclei and electrons which by and large shape the features and attributes of the molecules they compose. These molecules are defined by the arrangement of atoms connected by bonds according

to their valence, where valence refers to the ability of an atom to bond or interact with other atoms. Specifically, a covalent bond between two atoms is the result of a pair of electrons being shared by the two atoms. Given two atoms, there are four types of covalent bonds that may occur: i) a single bond is present when one pair of electrons is shared, ii) a double bond occurs when two pairs of electrons are shared, iii) similarly a triple bond is present when three pairs of electrons are shared, and iv) quadruple bonds occur when four pairs of electrons are shared. Quadruple bonds are possible only by using orbitals whereby the shared atoms form clouds above and below the plane where the bonded atom nuclei are located. Aromatic bonds occur when more than two atoms share electrons. Groups of bonded atoms are categorized based on their characteristics and behaviour during chemical reactions. These groups are referred to as functional groups. These functional groups play an essential role in producing chemical compounds desired by chemists. Furthermore, molecules are categorized into chemical classes based on the functional groups that they possess.

The process in which molecules transform to produce a desired compound is known as a chemical reaction. Chemical reactions are characterized by two main components: i) the reactants (functional groups) required for the chemical reaction to be applicable, and ii) the product(s) or outcome as a result of applying the reaction. The product(s) of the reaction consist only of the atoms that compose the reactants since atoms are not removed or added during the reaction. In other words, the atoms involved during the reaction do not change. However, the bonds that link the atoms as part of the reactants are modified. Bonds that were present previously in the reactants may be removed, remain the same, or new bonds may be formed as a result of the reaction.

Molecules are commonly illustrated as graphs whereby atoms are represented by the vertices and bonds are represented as edges of the graph. Intuitively, given two atom: i) a single bonds is represented by one single edge, ii) a double bond is shown as two parallel edges, and iii) a triple bond is illustrated by three parallel edges. Aromatic bonds between more than two atoms are represented by a ring consisting of straight line segments adjacent to a rounded line connecting the atoms. Hence, chemical reactions can be represented conveniently whereby the reactants required are present on the left hand side of an arrow (separated by "+"), and the resulting product(s) is shown on the right hand side of the arrow (separated by "+"). Figure 3.1 illustrates the reactants

(*acyl chloride* and *amine*) and product molecules that characterize the *AmideSynthesisFromAcidChloride* chemical reaction. An *acyl chloride* molecule is an organic compound consisting of a carbon, oxygen, and chlorine atoms, whereby the carbon atom is attached to a hydrocarbon molecule (i.e., organic compound containing only hydrogen and carbon atoms). Whereas an *amine* molecule is another compound which consists of a nitrogen atom that is attached to other substituents, often consisting of hydrogen atoms. Note that the numbers associated with each atom are given arbitrarily but maintained to explicitly illustrate the effect of the reaction on each atom. For instance, Figure 3.1 shows that the single bond between the $1^{st}$ atom, a chlorine atom, and the $2^{nd}$ atom, a carbon atom, exists before the reaction. However, this bond breaks after the reaction so that the $1^{st}$ atom forms a new single bond with the $8^{th}$ atom, a hydrogen atom, and the $2^{nd}$ atom forms a new single bond with the $5^{th}$ atom, a nitrogen atom. Since the atom numbers do not change, it is easy to compute the changes that occur in the bonds between atoms.

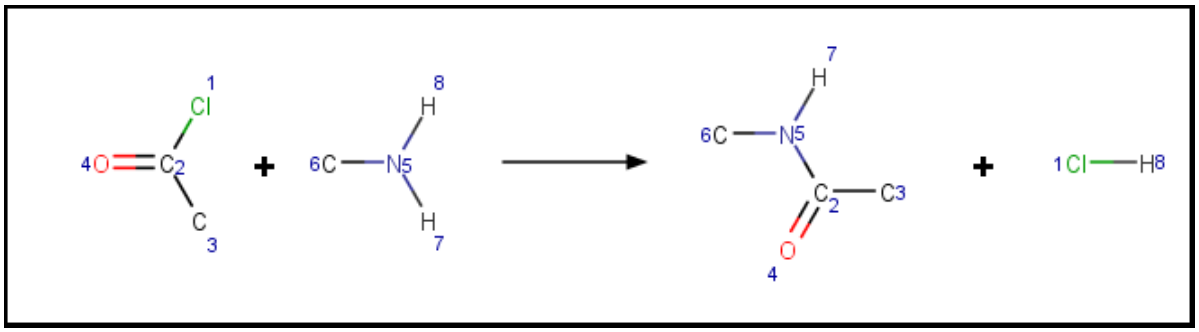



FIGURE 3.1: Illustrates the *AmideSynthesisFromAcidChloride* chemical reaction

### 3.1.2 Organic Chemistry Synthesis

Organic chemistry synthesis is a challenging field in organic chemistry. The focus of this study is on organic material whereby chemical reactions are applied on a given set of molecules in order to produce the desired target molecule(s). Organic synthesis is highly relevant and valuable in reality. Applications of this field include the discovery of synthetic routes for new molecules (possibly for pharmaceutical use) or to produce new materials with desirable properties. Additionally, organic synthesis can be used to identify new pathways to construct familiar compounds that in turn minimizes the cost or difficulty of the procedure. The study of organic chemistry

synthesis is extensive and has resulted in forming large commercially available knowledge bases that consist of chemical compounds and reactions. Hence, given a large number of chemical reactions and molecules available, identifying the chemical reactions required in order to produce the target molecule(s) is a challenging task. In particular, the number of achievable compounds may become exponentially large in the worst possible case. As a result, the effort required to produce and isolate the desired target molecule(s) is extensive. To simplify our research, we ignore common metrics associated with reactions such as yield (i.e., the percentage of useful products out of the total product produced) and the cost of materials and reactions (i.e., some reactions require expensive catalyzers and special equipment).

## 3.2 Organic Chemistry Synthesis in Planning

The challenge of solving organic synthesis problems creates an incentive to automate the procedure by employing computers to solve problems within this domain. The research field concerned with using computers in junction with the field of chemistry is known as Cheminformatics. In particular, this area is especially practical in the maintenance, retrieval and analysis of information regarding chemical compounds. Recently, Szymkuc et al. [67] have successfully made significant progress with regards to automating the organic synthesis procedure. However, their techniques used in problem solving are specific to the organic synthesis domain. Whereas we develop a domain-independent planner capable of solving problems in general that also include problems from the organic chemistry synthesis domain. In order to facilitate the process in which experts and trained chemists use to solve organic synthesis problems, techniques developed for efficient planning can be applied on this domain. Furthermore, organic synthesis is a challenging field that has real-world applications. In turn, the limitations of classical planning are made more prominent given that organic synthesis is considered as a planning task.

### 3.2.1 Representation of the Chemistry Domain in Situation Calculus

Our colleagues, namely Arman Masoumi and Mikhail Soutchanski, are the first research group to represent chemical molecules and reactions in situation calculus [45] as well as formulate organic synthesis as a planning task in AI [46]. In their work, a knowledge representation and

reasoning (KRR) framework is developed for representing and reasoning about the chemistry synthesis domain. Their KRR framework is based on situation calculus (SC) as discussed in Chapter 2. SC provides a logical formalization used to express actions and their effects in a dynamic environment. As a result, organic synthesis can be expressed as a planning problem whereby the task is to produce the sequence of actions that manipulate the molecules in the initial state to produce the molecule(s) in the desired goal state.

With regards to organic synthesis, we maintain an identical representation of molecules as in [44], and differ in how the chemical reactions are encoded. Atoms are expressed in SC as objects using situation-independent predicates based on the atom chemical name in Mendeleev's periodic table. For instance, *nitrogen(N1)*, defines the object *N1* to symbolize a nitrogen atom. Bonds are expressed in SC as situation-dependent relational fluents. These fluents may become true or false according to the effects of the actions applied as part of the given situation. Respective to the four types of chemical bonds, *bond(X,Y,S)*, *doubleBond(X,Y,S)*, *tripleBond(X,Y,S)*, and *aromaticBond(X,Y,S)*, are the fluents used to denote them in SC. Each fluent contains three parameters, $X$, $Y$, and $S$, to signify that atom $X$ is bonded with atom $Y$ in situation $S$, whereby the type of bond is expressed by the name of the predicate. Hence, in the initial situation $S_0$ as specified by the BAT's initial theory $D_{S0}$, the initial molecules are expressed using these predicates, fluents, logical connectors and quantifiers. For example, an ammonia molecule in the initial situation $S_0$ is represented in SC as:

$$
\begin{aligned}
&\textit{nitrogen(N)} \land \textit{hydrogen(H1)} \land \textit{hydrogen(H2)} \land \textit{hydrogen(H3)} \land \\
&\quad \textit{H1} \neq \textit{H2} \land \textit{H1} \neq \textit{H3} \land \textit{H2} \neq \textit{H3} \land \\
&\qquad \forall x.\ \textit{bond(x,N,}S_0\textit{)} \rightarrow (x{=}\textit{H1} \lor x{=}\textit{H2} \lor x{=}\textit{H3}) \land \\
&\qquad\quad \forall y\ (\ \textit{bond(H1,}\ y,\ S_0\textit{)} \rightarrow y{=}\textit{N}\ ) \land \forall y\ (\ \textit{bond(H2,}\ y,\ S_0\textit{)} \rightarrow y{=}\textit{N}\ ) \land \forall y\ (\ \textit{bond(H3,}\ y,\ S_0\textit{)} \rightarrow y{=}\textit{N}\ ).
\end{aligned}
$$

This formula establishes that four objects, *N*, *H1*, *H2*, and *H3*, exist as one nitrogen and three distinct hydrogen atoms in the initial situation respectively. Additionally, the formula asserts that the nitrogen atom is only bonded to the three hydrogen atoms by a single bond and not to any other atom. Similarly, the three hydrogen atoms are only bonded to the nitrogen atom by a single bond and not to any other atom.

For our purposes, chemical reactions are expressed in SC similarly to the representation as stated in [44]. This macro approach expresses each chemical reaction as an action term in SC. Each action is defined by its precondition axiom (PA) and successor state axioms (SSAs). These two key components describe the conditions that are necessary for an action to become applicable, and the resulting effects that occur upon applying the action. For instance, recall that Figure 3.1 demonstrates the requirements and the effects of the *AmideSynthesisFromAcidChloride* chemical reaction. In SC, the PA for the *AmideSynthesisFromAcidChloride* reaction specifies that it is applicable if and only if an *acylChloride* molecule and an *amine* molecule are present in the given situation. The arguments in the action's predicate only mention the atoms that are affected by the action as shown:

$$Poss(\ amideSynthesisFromAcidChloride(C2, Cl1, H8, N5), s\ ) \leftrightarrow$$
$$acylChloride(C2,C3,O4,Cl1, s) \wedge$$
$$amine(N5, H7, H8, C6, s).$$

where *acylChloride(C2,C3,O4,Cl1, s)* and *amine(N5, H7, H8, C6, s)* are abbreviations constructed out of primitive fluents, predicates, and possibly, simpler abbreviations.

Although we follow the same formalism as the work produced in [44], our encoding differs in the representation and identification of the preconditions for actions. In [44], the PA for each action is encoded as a long conjunction of literals. In turn, the instantiation of literals is solved as a constraint satisfaction problem (CSP). The complexities of solving CSPs are well known and have been a difficult research issue for some time [69]. Specifically for the organic synthesis domain, in the previous implementation [44], the task of computing instantiations that satisfy the conjunction of literals is solved through exhaustive search and backtracking. Ultimately, this limitation creates a major bottleneck for solving planning problems in the organic synthesis domain.

Subsequently, the SSA for the *AmideSynthesisFromAcidChloride* reaction specifies that four atoms are affected as a result of applying the reaction. Specifically, the single bond between *{Cl1, C2}* and the single bond between *{N5, H8}* are disconnected. Additionally, a new single bond between *{N5, C2}* and another single bond *{Cl1, H8}* are formed. All other bonds remain the same as shown:

$bond(x, y, do(a,s)) \leftrightarrow$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(x, t1, t2, y)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(y, t1, t2, x)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(t1, x, y, t2)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(t1, y, x, t2)\ ) \vee$

$bond(x,y,s) \wedge \neg$

$(\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(x, y, t1, t2)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(y, x, t1, t2)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(t1, t2, x, y)\ ) \vee$

$\exists\, t1,t2(\ a = amideSynthesisFromAcidChloride(t1, t2, y, x)\ )).$

Notice that for each bond, the action is repeated twice since the bonds are symmetric relations.

Our work builds upon this representation of organic synthesis as a planning problem. We contribute by correcting the chemical reactions and problem instances provided by collaborating with experts in chemistry and by consulting verified online resources. Furthermore, we resolve the lack of efficiency due to the CSP when grounding action arguments. Specifically, in our work, each precondition (functional group) of an action is represented as a graph to be assessed by an efficient implementation of sub-graph isomorphism against the current state. Whereby, the current state represents all bonds existing in the current situation. Our encoding of functional groups as graphs and the application of sub-graph isomorphism are explained in the following sections of this chapter.

### 3.2.2 Representation of Functional Groups as Graphs

The nature of the organic synthesis domain is highly complex. Specifically, when solving a problem, the number of compounds given initially could be limitless in theory. As a result, choosing the correct compound to apply in the reactions becomes challenging. Additionally, the identification of applicable reactions for a given state poses a challenge. In order for a reaction to become applicable, the reactants must be present in the given state. The identification of the reactants is difficult given that the previous encoding of preconditions consisted of a long conjunction of literals. Since the instantiation of arguments for these literals was considered as a CSP, attempts

to produce a solution were very limited by this approach. The third main difficulty is that the correct preconditions must be chosen and in the correct configuration. Given that some molecules are symmetric in 2-dimensional space, matching of such a molecule must take its orientation into account since the result of applying the action will only affect certain atoms in the molecule. However, all variations of this molecule must be considered in order to satisfy the conditions for subsequent actions correctly. The fourth main difficulty is the inherent combinatorics in this domain. Given that such a large number of actions exist in the domain, the preconditions will be assessed frequently. In turn, an efficient approach for assessing the preconditions of actions is necessary. The fifth main difficulty involves the evaluation of whether the goal molecule is achieved since the target is defined as a long conjunction of bonds between the atoms, and therefore, computing this conjunction takes time. Hence, the domain poses several difficulties which are managed inefficiently by relying on the CSP approach to identify the existence of particular molecule within the given state.

To manage these difficulties, we consider a graph-based representation of the molecules. Given that the solver will spend a significant amount of time in assessing these graphs, the representation should be chosen to use time and memory efficiently. Furthermore, this representation should be intuitive and easily legible for the user. As a result, the decision was made to represent molecules using adjacency lists. Given that all atoms in the molecule must be traversed to ensure isomorphism, adjacency lists are no less efficient than a tree type data structure for instance given that all elements must be assessed. Unlike an adjacency matrix, the adjacency list is compact in size by maintaining only one list of the associated nodes. In contrast, the adjacency matrix and similar structures need to reserve multiple lists that would take up additional space unnecessarily. Syntactically, adjacency lists are easily traversable and clearly legible to the user. As a result, each adjacency list is comprised of terms, where each term includes the following elements: i) the atom, ii) the atom type, and iii) the list of adjacent atoms that are linked by a chemical bond. The following example illustrates an adjacency list which represents the water molecule. The water molecule consists of one oxygen atom, *O1*, that is linked by a single bond to each of the two distinct hydrogen atoms, *H1*, and *H2*. Reciprocally, each hydrogen atom is linked by a single bond to the oxygen atom.

```
/*water*/
water(O1,H1,H2,
  [
    atom(O1,oxygen,
      [
        bond(H1, hydrogen, single),
        bond(H2, hydrogen, single)
      ]
    ),
    atom(H1,hydrogen,[bond(O1, oxygen, single)]),
    atom(H2,hydrogen,[bond(O1, oxygen, single)])
  ]).
```

In general, the functional groups are represented as adjacency lists, whereby the adjacency lists are composed of terms of atoms. However, for certain functional groups, they are defined using previous definitions of other functional groups. For instance, the *sodiumDichromate* functional group is composed of two *chromate* groups that are linked by an oxygen atom through a single bond. Thus, in order to reduce redundancy and make use of previously encoded functional groups, in these cases the adjacency lists are nested to include terms of molecules as well. Subsequently, we are able to apply an efficient sub-graph isomorphism algorithm to resolve the issue with the CSP in order to identify the applicability of actions int the given state of molecules. The following section reviews our implementation of sub-graph isomorphism in detail.

## 3.3 Sub-graph Isomorphism

With regards to pattern analysis and pattern recognition in artificial intelligence and machine learning, graphs are commonly used to represent and maintain large collections of data conveniently. Generally, the graphs consist of nodes and edges which are possibly directed. Upon capturing some phenomena, it is decomposed into parts which are then associated with nodes and edges in a meaningful way. Nodes contain semantic information regarding the phenomena, and the edges reflect the relationship between them. In turn, this decomposition allows for the ability to compare the graphs for similarities by recognizing common patterns among them. The

computational problem concerned with identifying a mapping of nodes from a given guest (pattern) graph to a given host (target) graph, if such a mapping exists, is referred to as sub-graph isomorphism. The sub-graph isomorphism problem is known to be NP-complete. In order to perform isomorphism techniques on large complex graphs, an efficient algorithm is required. Ideally, the algorithm should be applicable in general and not suited to perform well only on a particular domain.

A domain-independent sub-graph isomorphism algorithm was introduced by Ullmann [68] in 1976. The Ullmann algorithm became most commonly used as it employed an efficient backtracking procedure that reduced the size of the search space. Subsequent (sub)graph isomorphism algorithms gauged their efficiency by comparing their results with the Ullmann algorithm. More recently, a significant improvement over the Ullmann algorithm was demonstrated by the VF algorithm produced by Cordella et al. [7]. The VF algorithm allowed for efficient (sub)graph isomorphism for graphs involving larger than 1000 nodes. The VF algorithm was later revised to produce the VF2 algorithm [8] by reorganizing the data structures during the search and incorporating additional feasibility rules. These revisions further improved the memory efficiency of the (sub)graph isomorphism algorithm. Experiments performed by Kotthoff et al. [36] involving a number of state-of-the-art sub-graph isomorphism algorithms demonstrate that the VF2 algorithm is superior. Furthermore, work performed by McCreesh et al. [49] compare (sub)graph isomorphism algorithms on difficult random instances of graphs. Throughout their investigation, it is found that the VF2 algorithm exhibits difficulty in matching graphs on particularly rare and extremely difficult instances. Additionally, further research demonstrates that the VF2 algorithm is used in chemistry in order to identify molecular substructures in large databases efficiently [13].

For our purposes, the VF2 algorithm is chosen to be implemented as it makes use of strict and comprehensive rules that quickly allows for the identification of isomorphism among graphs. Furthermore, it is shown that the VF2 algorithm can efficiently identify and extract molecular sub-graphs regarding chemical compounds from large databases efficiently. The following section outlines the cost-optimized sub-graph isomorphism algorithm based on the VF2 algorithm. As a result, this algorithm proved to be a potential candidate in reducing the excessive time spent in exhaustively assessing all combinations of atoms when determining if an action is applicable in a given state.

### 3.3.1 Sub-graph Isomorphism Overview

To determine if two graphs are isomorphic, the given graphs are labelled as the host graph, $H$, and the guest graph, $G$ respectively. Formally, isomorphism is assigned to produce an edge- and label-preserving bijection of nodes that compose the given graphs. In other words, the task is to identify if there can exist a mapping of all nodes within the guest graph onto a subset of nodes within the host graph whilst satisfying all of the rules that are imposed during this mapping. It is assumed that each graph consists of at least one node. Each node contains semantic information about the data that the graph represents. Similarly, the edges which connect the nodes within the graph may also be associated with additional information (i.e., direction or edge type).

The procedure begins by selecting a node, $g$, from $G$ and determining if a corresponding node, $h$, exists within $H$. If the candidates from each graph satisfy all necessary conditions, then a subsequent node, $g'$, that is adjacent to $g$, is chosen for the next iteration. The procedure gradually builds the cloud of vertices around $g$ and the corresponding cloud of vertices around $h$. This expansion procedure continues until $G$ has been completely traversed and all nodes within $G$ are mapped onto a subset of nodes within $H$. The verification process assesses candidate nodes by consulting two sets of rules: 1) the syntactic feasibility rules, and 2) the semantic feasibility rules.

The syntactic feasibility rules enforce the conditions which regard the structure of the graph. Given a guest candidate node, $g$, and host candidate node, $h$, the following three syntactic feasibility rules establish if the nodes are identical according to their arrangement:

A) For each distinct mapped node that is adjacent to $h$, there exists a distinct mapped node that is adjacent to $g$, and vice versa. In other words, this rules verifies that the nodes which were mapped previously remain to be mapped and nodes are not mapped repeatedly.

B) Identify *adj_h* to be the nodes in the host graph, not yet mapped, that are adjacent to the host candidate node as well as adjacent to nodes that were previously mapped in the host graph. Similarly, identify *adj_g* to be the nodes in the guest graph, not yet mapped, that are adjacent to the guest candidate node as well as adjacent to nodes that were previously mapped in the guest graph. Ensure that there are at least as many nodes in *adj_h* as there are in *adj_g*. To maintain

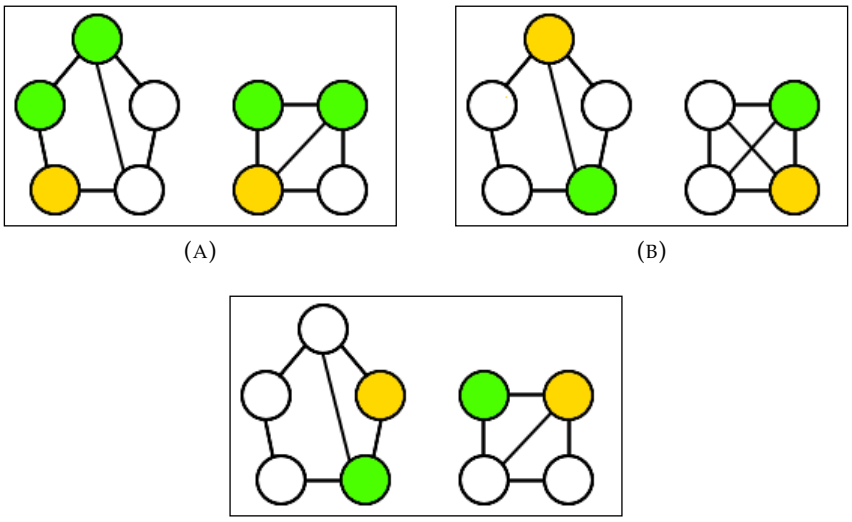
efficiency, this rule performs a brief and local look-ahead to ensure that the host candidate node has at least as many unmapped neighbours, that are adjacent to previously mapped nodes, as the guest candidate node.

C) Identify *adj_h'* to be the nodes in the host graph, not yet mapped, that are adjacent to the host candidate node, but not adjacent to nodes that were previously mapped in the host graph. Similarly, identify *adj_g'* to be the nodes in the guest graph, not yet mapped, that are adjacent to the guest candidate node, but not adjacent to nodes that were previously mapped in the guest graph. Ensure that there are at least as many nodes in *adj_h'* as there are in *adj_g'*. Similarly to the second syntactic rule, this rule performs a brief and extended look-ahead to ensure that the host candidate node has at least as many unmapped neighbours, that are not adjacent to previously mapped nodes, as the guest candidate node.

Figure 3.2 illustrates an unsuccessful attempt during the mapping procedure for each syntactic feasibility rule. Note that for each sub-figure, the graph shown on the left hand side is the host graph, $H$, and the graph shown on the right hand side is the guest graph, $G$. In both graphs, the nodes highlighted in green denote that they have been mapped. Whereas, nodes highlighted in yellow denote that they are potential candidates: $h$ within $H$, and $g$ within $G$ respectively. All other (white) nodes have not yet been considered in the mapping procedure.

In each case, the mapping is unsuccessful due to the failure of satisfying one of the syntactic feasibility rules. In sub-figure (A), the first syntactic rule is not satisfied since each distinct mapped node that is adjacent to $g$ does not correspond with a distinct mapped node that is adjacent to $h$. In sub-figure (B), failure to satisfy the second syntactic rule occurs since there are less nodes which are adjacent to $h$, also adjacent to nodes that were previously mapped in $H$, than nodes which are adjacent to $g$, also adjacent to nodes that were previously mapped in $G$. Sub-figure (C) demonstrates that the third syntactic rule is not satisfied because there are less nodes which are adjacent to $h$, and not adjacent to nodes that were previously mapped in $H$, than nodes which are adjacent to $g$, and not adjacent to nodes that were previously mapped in $G$.

The semantic feasibility rules enforce the conditions with regards to the context of information



(A) (B)

(C)

FIGURE 3.2: Illustrates cases A, B, and C, where the syntactic feasibility rules fail, respectively

contained within the graph. Given a guest candidate node, and host candidate node, the following three semantic feasibility rules establish if the nodes are identical according to the information that they maintain:

A) The candidate node within the guest graph and the candidate node within the host graph share the same type.

B) For each distinct edge type that is connecting to the candidate node within the guest graph, there exists an identical distinct edge type that is connecting to the candidate node within the host graph respectively.

C) For each distinct neighbour of the candidate node within the guest graph, there exists a distinct neighbour of the candidate node within the host graph that shares an identical type respectively.

If the syntactic and semantic rules are not satisfied for two candidate nodes, then the algorithm backtracks to choose a new host candidate node. If all possible host candidate nodes are

exhausted, then the algorithm returns to choose a new guest candidate node. This expansive procedure continues until all nodes within $G$ are found to correspond to identical nodes within $H$ respectively, whilst satisfying all conditions of the syntactic and semantic feasibility rules. Otherwise, the algorithm fails given that all nodes within the guest graph cannot be mapped to nodes within the host graph. Algorithm 3.1 illustrates the sub-graph isomorphism procedure.

**Algorithm 3.1:** isomorphism

**Input:** the host graph ($H$), and the guest graph ($G$).
**Output:** the list of corresponding mapped nodes from both graphs.
*listMapped* **is** []
1. **chooseG:**
2. **if** all nodes in $G$ are mapped to nodes in $H$, **then**
3. **return** *listMapped*
4. **else**
5. select a candidate node, $g$, from $G$
6. select a candidate node, $h$, from $H$
7. **map:**
8. Syntactic_Feasibility_Rules(g, h) % assess candidates nodes for syntactic feasibility
9. Semantic_Feasibility_Rules(g, h) % assess candidate nodes for semantic feasibility
10. **if** all feasibility rules are satisfied, **then**
11. append(*listMapped*, mapping(h,g)) % append the pair of corresponding nodes to the list of mapped nodes
12. adjacent(g, g') % produce the list of nodes, $g'$, that are adjacent to $g$
13. adjacent(h, h') % produce the list of nodes, $h'$, that are adjacent to $h$
14. proceed to *step 1* with $g'$ and $h'$
15. **else**
16. select a new candidate node, $h$, from $H$
17. **if** all candidate nodes from $H$ fail, **then**
18. select a new candidate node, $g$, from $G$
19. **if** all candidate nodes from $G$ fail, **then**
20. *listMapped* **is** []
21. **return** listMapped
22. proceed to *step 7* with $h$ and $g$ % since the feasibility rules must be assessed for the new candidates nodes

### 3.3.2 Sub-graph Isomorphism in Chemistry

In regards to the organic synthesis domain, the nodes and edges within a graph will contain information about the molecule that it represents. The semantic information held by nodes consist of the names of atomic elements (i.e., hydrogen, helium, oxygen, etc.). Furthermore, the types of edges in the graphs correspond to the type of chemical bond shared between two atoms: single, double, triple, and aromatic.

To further improve the efficiency of the algorithm, two enhancements are carried out. First, the nodes of the host and guest graphs are sorted such that the nodes which possesses the least number of edges are to be selected first in the mapping procedure. By selecting the node that is located in the least dense area of the graph earlier, the algorithm will be limited by the number of possible candidates in subsequent iterations. In turn, the procedure is more likely to choose the correct mapping of nodes at an earlier stage.

Second, during the mapping procedure, upon identifying a corresponding host node, $h$, for a given guest node, $g$, the subsequent guest graph nodes are chosen strategically. The nodes adjacent to $g$ are sorted by the frequency of occurrence according to the type of bond that links them to $g$. In other words, adjacent nodes to $g$ are chosen for further expansion of the procedure in the order of the type of edge that connects them to $g$. The adjacent nodes are sorted in descending order of occurrence as follows: triple bond, double bond, aromatic bond, and single bond. As a result, the procedure will eliminate incorrect potential candidates nodes earlier and prune potentially incorrect matchings in the future in significantly less time. These enhancements have shown to substantially improve the efficiency of the algorithm when identifying the applicability of actions. As a result, throughout our experiments, the isomorphism algorithm has been shown to identify the preconditions of actions immediately and ultimately alleviate the difficulty posed by assessing preconditions of actions as a CSP.

## 3.4 DFIDS Solver and BinTree Solver

A brief description of the two solvers applied on the organic chemistry synthesis benchmark is given in this section. Throughout the planning procedure, the amount of time and memory resources provided must be taken into account. The given DFIDS solver maintains the state by

preserving a situation argument for all fluents. The validity of these fluents is assessed recursively given the situation associated with each fluent. Whereby, the situation is composed of all actions that were applied from the initial state. For domains, such as organic chemistry synthesis, that involve a significant number of fluents within a problem instance, the method used to assess each fluent in a linear search time given the situation was hypothesized to be inefficient.

Consequently, a new approach of maintaining the state in a data structure throughout the planning procedure was developed to produce the *BinTree* solver. The solver represents the initial and subsequent states as a balanced binary tree. The number of nodes in this tree is calculated as the number of distinct objects that occur as the first argument in the fluents of the initial state. The fluents are grouped and placed within the corresponding node. The left and right sub-trees include an approximately equal number of nodes. An example of the binary tree representation is illustrated in Figure 3.3 for the *blocksWorld* example shown in Figure 2.6 whereby the initial state is {Clear(C, $S_0$), On(C, B, $S_0$), On(B, A, $S_0$), On(A, Table, $S_0$)}.

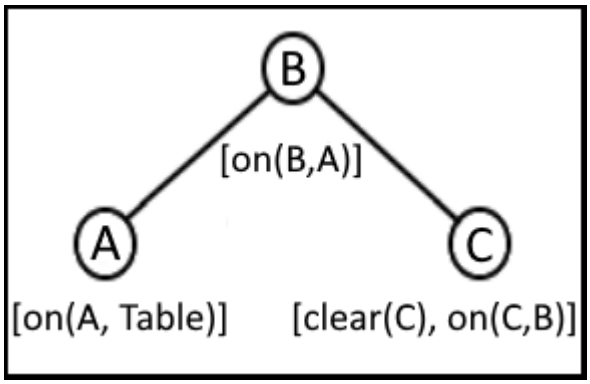



FIGURE 3.3: Illustrates the binary tree for the *blocksWorld* example in Figure 2.6

This type of tree allows for the most ideal relation between the number of nodes and the height of the tree. Once the tree is constructed, it maintains its balance as the shape remains unchanged throughout the procedure since nodes are not added or removed. In turn, it provides reasonably fast access, insertion, and deletion of the literals which compose the state. To conform with this representation, the positive and negative effects of actions are explicitly encoded without a situation argument. Specifically, a positive effect is a fluent that becomes true as a result of applying an action. This fluent is inserted into the respective node of the binary tree. Correspondingly, a negative effect is a fluent that becomes false as a result of applying an action. This fluent is deleted

from the corresponding node of the binary tree. Although the worst case complexity of verifying or removing a fluent from the list of $n$ literals in a given node is O(n), the lists are generally small in size. As a result, the worst case complexity of verifying (inserting or removing) a fluent in a given state represented in a binary tree is O($log_2(n)$), where $n$ is the number of constants mentioned as the $1^{st}$ argument in the fluents in the initial situation. As an added feature, this encoding enables the use of heuristics by representing the effects of actions explicitly without a situation argument.

## 3.5 Chemistry Data Set

Originally, Heifets and Jurisica [22, 21] developed a benchmark set of 20 organic synthesis problems based on undergraduate organic chemistry examination questions from the MIT using a specialized chem-informatics format. Subsequently, the data set was amended by Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski [45, 44] to be used in domain-independent planning. To ensure the chemical accuracy of the data set, we have corrected the previous encoding (summer 2016) by consulting with professors in chemistry at Ryerson University, Dr. Anne Johnson, Dr. Russell Viirre, Dr. Sharonna Greenberg, and Dr. Andrew McWilliams. The problems vary in difficulty with regards to the number of objects they possess, from as little as 33 atoms up to 135 atoms. The problems also vary with regards to the complexity and length of the solution required, from 3 steps to 13 steps.

The reactions were developed using *MarvinSketch*, a software (developed by the company ChemAxon) with a GUI editor used to represent chemical reactions. The reactions were stored in the *RXN* format (version *V2000*). The provided Python RXN translator was used to produce the output in PDDL. The PDDL encoding of the chemical reactions and problem instances were submitted to the IPC-2018, for which we received the *Outstanding Domain Submission Award*.

For our purposes, the set of reactions was store das the *MRV* format that is based on XML. MRV supports the representation of content-rich chemical structures and presents them in a format that can conveniently be parsed and processed. Specifically, MRV is a type of Chemical Markup Language (CML) that is expressed in the XML format. We then developed the Java *MRVToBinTree* translator to convert the MRV reactions to Prolog actions with preconditions as functional groups in order to apply the BinTree solver on the corrected data set. In order to produce the correct

encoding of the chemical reactions, the functional groups required by the reactions must be identified correctly. Ideally this process should be automated to avoid additional effort and human error. Further research was conducted to identify these functional groups in CML [65]. However after some consideration, we took advantage of the Java-to-Prolog interface [12] and relied on our Prolog sub-graph isomorphism implementation to correctly label the functional groups and produce the action encodings correctly. As a result, the MRVToBinTree translator produced the actions in Prolog to be used in accordance with the binary tree representation of states.

## 3.6 Experimental Results

To gauge the improvement of efficiency gained by the BinTree DFID search planner over the standard DFID search planner, both solvers were applied on the 20 benchmark problems. Table 3.1 illustrates the results produced by both solvers. It is shown that both solvers are able to produce the optimal plan for 7 out of 20 problems. Each instance was given a time limit of one hour, although for the seven solvable problems, both solvers were generally able to produce a solution in less than two minutes. Results were produced under the following specifications: i5, single CPU core, @2.50GHz, 2GB RAM. Since the solvers relied on uninformed depth-first search, memory was not an issue, and more benchmark problems could be solved given enough time.

## 3.7 Analysis

For the problems that could be solved, the results produced by both solvers are nearly identical whereby the BinTree solver produces a solution in less time in general. The binary tree representation and maintenance of the state showed some improvement over the DFIDS solver which updates the state recursively for each fluent given the situation. Given that the planners could only solve a subset of the problems, the hypothesized gain in performance by the BinTree solver is not consistently shown. For instance, the BinTree solver only demonstrates considerable performance over the standard DFIDS solver for *problem 10* since the number of objects in this problem are significantly large. Also, the actions used to produce a solution required a significant number of preconditions. Hence, this gain in performance must be due to the way in which the BinTree

| Problem ID | Number of Objects | Optimal Number of Steps Required | DFIDS Solver | BinTree Solver |
|---|---|---|---|---|
| p1 | 119 | 5 | na | na |
| p2 | 44 | 4 | na | na |
| p3 | 70 | 5 | 91.89 | 82.83 |
| p4 | 76 | 8 | na | na |
| p5 | 48 | 3 | 24.88 | 18.14 |
| p6 | 33 | 7 | 58.14 | 51.2 |
| p7 | 78 | 9 | na | na |
| p8 | 119 | 6 | na | na |
| p9 | 106 | 7 | na | na |
| p10 | 72 | 3 | 143.56 | 105.67 |
| p11 | 76 | 13 | na | na |
| p12 | 95 | 10 | na | na |
| p13 | 70 | 3 | 51.83 | 47.95 |
| p14 | 73 | 5 | 79.80 | 81.7 |
| p15 | 99 | 13 | na | na |
| p16 | 53 | 4 | na | na |
| p17 | 96 | 5 | na | na |
| p18 | 135 | 9 | na | na |
| p19 | 59 | 5 | 38.34 | 42.81 |
| p20 | 60 | 10 | na | na |

TABLE 3.1: Comparative results produced by BinTree DFIDS and standard DFIDS on the Organic Chemistry Synthesis benchmark

solver represents and maintains the state since this is the only difference between the solvers. If both solvers could solve the remaining problems, then this trend would be evident given by the results.

There are two two main limiting factors that prevent the solvers from producing a solution for the remaining problems of the benchmark. The first factor is the number of objects involved within the problem instance. The number of objects add to the complexity by allowing more actions to become applicable. Consequently, the increase in the number of objects in a given problem leads to an increase in the number of actions that can be applied. The second limiting factor is the number of steps required to solve the problem. For each additional step required to solve a problem, the number of choices to explore increases exponentially at each depth in the worst possible case.

Nevertheless, results produced by the BinTree solver have shown significant progress with regards to previous attempts to solve this benchmark. Heifets and Jurisica [22] relied on an IBM

super-computer, whereby each problem was given 8GB of RAM and up to 6 hours of CPU time to produce a solution. As a result, only a subset of problems could be solved given these resources (i.e., problems 16 - 20 could not be solved). Furthermore, Matloob and Soutchanski [47] have shown how state-of-the-art planners are able to encode only a portion of the benchmark problems due to the grounding issue. They also explore an approach to alleviate this issue by splitting each action into sub-actions with a shorter interface by making use of action schema splitting syntactic transformations. However, it was shown that only a small number of problems could be solved in their attempt. In any case, this benchmark remains to pose a significant challenge for modern AI planners.

## 3.8 BinTree solver Improvement Attempt

Effort was spent to speed up the procedure by employing best-first search within the BinTree solver. The heuristic in this regard was calculated to be the minimum number of steps required to produce the goal without taking in to account the negative effects of actions. At each step of the planning procedure, the neighbouring states are produced. For each resulting state, all applicable actions are applied without taking the negative effects of actions in to account. The procedure is repeated until the goal state is satisfied. The major bottleneck of this approach is that the states of atoms and molecules become increasingly interconnected as bonds are consequently made true without making the previously existing bonds false. The resulting graph in the relaxed state becomes exceedingly dense. While the isomorphism algorithm is efficient, it suffers on the relaxed state since it must account for all functional groups that exist in the resulting state. Recall that the isomorphism algorithm is used to identify applicable actions in the resulting state. Also the algorithm is used to identify if the goal state is reached. Both areas take a significant amount of time that result in the solver not being able to produce a solution better than the depth first approach without incorporating heuristics.

# Chapter 4

# Implementation and Experiments

This chapter discusses the methodology used to develop the RBHP algorithm. An overview of the planner's operation is presented. This overview includes the description of how the planner makes use of previously existing algorithms while avoiding their inefficiencies. In particular, the implementation of RBHP is illustrated in detail. Next, we present the data set upon which the algorithm is evaluated, followed by the experimental results produced by RBHP. Moreover, competitive results produced by RBHP against the top three scoring planners from the latest (9th) International Planning Competition are presented. Lastly, the results are analysed to formulate the reasons for the gain and lack in performance displayed by RBHP.

## 4.1 RBHP Overview

RBHP is a forward search planner that constructs the heuristic backwards only once using goal-regression. Moreover, the planner applies domain-independent heuristics, lifted to first-order logic, for best-first search. The planner operates in four phases: i) Preliminary Phase, ii) Regression Phase, iii) Heuristic Construction Phase, and iv) Search Phase.

During the preliminary phase, the initial state is constructed as a binary tree. Given the initial state as a set of literals, RBHP organizes the set alphabetically according to the object in the first argument within each literal. Then the sorted literals are placed into groups such that literals which share identical objects in the first argument are grouped together. Literals that possess distinct objects in the first argument are placed in distinct groups. Then each group of literals is inserted into the corresponding node within the binary tree. The current state is represented as a

binary tree and maintained by updating the previous state after the execution of each action. The goal state, given as a list, is maintained throughout the planning process.

Throughout the regression phase, RBHP focuses on objects mentioned in the goal formula by performing goal regression. The goal state is given as a list of goal literals. Each goal literal is regressed as in Unpop. Specifically, given a goal that is not satisfied in the initial state, all actions that achieve the goal are identified. To regress through these actions, their arguments must be grounded. For each action, the arguments within its preconditions are matched with the arguments existing in the goal literal achieved by the action. The remaining preconditions with variable arguments are matched with literals that are satisfied in the initial state. Preconditions that are not true in the initial state are grounded using all variations of objects in the initial state. Note that this can potentially produce irrelevant ground actions if the initial state includes redundant objects. If an action still contains variable arguments, then it is discarded as it cannot attain the goal literal.

Upon successfully producing the grounded actions, regression is applied through all actions in order to identify the sequence of actions that can achieve the goal greedily with the least amount of effort. Note that the effort is defined as the number of actions required to achieve the goals from the initial state. The preconditions of each action are assessed to identify if they are satisfied in the initial state. If the preconditions of an action are satisfied in the initial state, then a solution is found to satisfy the original goal literal. If the preconditions of an action are not satisfied in the initial state, then the preconditions are treated as new goal literals and are further regressed by performing depth-first search. Regression is applied through each action until an action is found such that its precondition are true in the initial state.

However, unlike Unpop, RBHP performs the regression phase only once for the entire planning procedure since the regression graph is unique. Whereas, Unpop performs the regression-match graph at every step during the planning procedure in order to produce the heuristic estimate to assist in guiding the search. The reapplication of the regression procedure does not provide any additional information. Hence, RBHP stores the information obtained from applying regression into the heuristic table in order to avoid the excessive computational effort as performed by Unpop.

Furthermore, the regression procedure is optimized with regards to time and memory resources.

Specifically, memoization is used to cache the information obtained about the goals which were previously regressed successfully. In this way, regression is not performed redundantly. Furthermore, the information that was previously obtained allows for the identification of cycles. Recognizing a cycle in which a goal is required to achieve itself is essential to avoid unnecessary steps of regression.

Once all goal literals are regressed successfully, the regression phase returns the regression graph. Mainly, the resulting regression graph details how each goal can be satisfied from the initial state in order to populate the heuristic table used during the search phase. This graph consists of alternating proposition and action layers which resembles the planning graphs produced by GRAPHPLAN. Furthermore, the regression graph can be used to efficiently produce the heuristic estimate that is identical to the reachability heuristic produced by GRAPHPLAN. Note, the reachability heuristic computed in the regression phase differs from the heuristic estimated computed during the search phase. During the regression phase, the reachability heuristic is used to assign a group of goal literals with the estimated minimum effort required to satisfy the group together. Recall that the effort is defined as the number of actions required to achieve the goals from the initial state. When regression is applied through several actions that achieve a goal as a positive effect, the effort required in order to achieve the goal must be computed accurately in order to identify the path which achieves the goal greedily. The reachability heuristic is admissible and more accurate than using the additive or maximum estimate required to satisfy the goals [5]. Once the effort required to satisfy a goal by applying regression through an action is estimated, regression through the subsequent actions is performed with better accuracy. In turn, regression through subsequent actions is terminated when the number of regression steps exceeds the current minimum effort. Hence, the reachability heuristic plays an essential role in accurately identifying the effort required to achieve a group of goals.

Unlike GRAPHPLAN, RBHP computes the reachability heuristic estimate without the need of producing the projection tree required by the planning graphs. Constructing the planning graphs take a considerable amount of time and memory resources.

Once, the graph is successfully constructed, an extraction procedure is performed. For each goal literal within the goal state, the sequence of actions that is necessary to satisfy the literal from the initial state is obtained. Note that the sequences of actions extracted from the regression graph

may not be complete. In particular, they are not guaranteed to produce the goal from the initial state since the sequence only consists of necessary actions that are required to produce the goal literal. The sequence is not guaranteed to include sufficient actions that may also be required to produce the goal. For instance, given the example initial and goal states from the *blocksWorld* domain displayed in Figure 4.1, the necessary actions required to satisfy the goal (i.e., {*on(a,b)*}) are: {*unstack(c,a)*, *pickup(a)*, *stack(a,b)*}. Note however, that this list does not include *putdown(c)*.

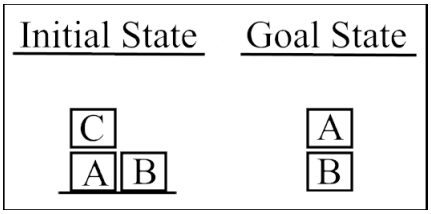



FIGURE 4.1: Example from *blocksWorld* describes unnecessary and sufficient actions

Since the action sequence does not contain all of the actions required to produce the goal, the heuristic table will not as precisely be able to guide the search. However, the consequence of missing sufficient actions in the action sequence is not especially significant. Given a state where a sufficient action must be performed, the planner will generally choose the correct action. If an incorrect action is chosen, the search will continue along an invalid path that will take more steps than necessary to produce the goal state as a whole. The heuristic estimate produced from subsequent states will deter the search from performing excessive actions. Hence, the correct action will be chosen given that the subsequent actions will lead the search closer to the goal state.

Similarly, a second scenario arises which involves action sequences that lack all necessary actions to achieve the goal state. Some goal states will involve goals that are already true in the initial state. However, it may be necessary to make these goals false temporarily in order to make some other goal true in the resulting goal state. For these instances, the resulting regression graph (produced by the regression phase), will not hold any information for these goals since they are already satisfied in the initial state. However, during search, the heuristic estimate of achieving these goals is zero since they are originally satisfied in the initial state. Hence, the planner will attempt to regain the goal that was made false since achieving the goal will guide the search closer to the goal state.

Once the sequences of actions that may produce the goals are extracted, each sequence is then used to populate the heuristic table to provide estimates during forward A* search. The heuristic table is generated during the heuristic construction phase. Each action sequence is assessed in the direction from the goal state towards the initial state so that the actions which are closer to producing the goal literals are reviewed earliest. The process continues until the final action in the sequence (i.e., the action applicable in the initial state) is reviewed. Similar to the GRT, the heuristic table is populated by a list of facts. The facts are attributed with three elements: i) a literal, ii) its respective distance from the goal, and iii) its set of related facts. The literal corresponding to each fact belongs to the set of negative effects of actions within an action sequence. The negative effects of an action are included since they depict the information necessary in order to execute said action. The respective distance to the goal corresponds to the index of the action in the sequence. Generally, the related facts of a literal consist of: i) all other negative effects of the action, ii) the preconditions of the action which are not deleted as a result of applying the action (i.e. non-delete preconditions), iii) the positive effects' set of related facts, and iv) the non-delete precondition's set of related facts. The resulting heuristic table includes a tentative solution path for the goal state from the initial state. In other words, the table consists of literals that must be achieved in order to satisfy the goal state from the initial state.

Recall that for the GRT, the table is constructed by applying inverted actions. The GRT is built once, in the pre-processing phase. The GRT essentially performs search during this phase by applying inverted actions, starting from the goal state. However, GRT suffers from a significant limitation. Generally, the goal state of a planning task is incomplete. In other words, the goal does not make a reference to all objects involved in the initial state. Since the goal states may be incomplete, inverted actions cannot be applied unless the goal states are enhanced. The enhancement includes new facts that do not contradict the original goals. A supposition is made for all objects that are not involved in the goal state. Literals that may involve these objects are made true for all possible scenarios. For example, the goal state of an instance from the *blocksWorld* domain may not state where the irrelevant blocks may be placed. In order to apply inverted actions, the blocks that are not part of the goal state are explicitly stated to exist in all possible configurations. Hence, GRT suffers from an additional operating cost. This overhead imposes additional effort with regards to time and memory resources [61]. However, RBHP solves for goals by applying regression on the

(incomplete) goal state, as opposed to applying inverted actions. As a result, RBHP can produce solutions for the problems that involve an incomplete goal state while avoiding this overhead.

RBHP proceeds with the planning phase by employing best-first search. The planner relies on an efficient situation-calculus based implementation of the A* algorithm (developed by my graduate advisor, Dr. Mikhail Soutchanski, 2014). The implementation uses a priority queue (developed by David Poole, 1996) to maintain plan prefixes (i.e., initial sequences of actions) throughout the planning phase. This planner is domain-independent in a sense that it can work with any planning problem. However, this planner can solve a given instance successfully only if a provided heuristic can correctly evaluate the progress towards a goal state. Maintaining plan prefixes instead of states within the search frontier reduces the burden on memory resources. The prefix with the highest priority (i.e., lowest estimated distance to the goal state) is selected. Then, the planner identifies all feasible actions and computes their resulting states represented as binary trees, respectively. Subsequently, the heuristic table is consulted in order to evaluate each resulting state in the frontier with a heuristic estimate. The evaluated plan prefixes are inserted into the queue according to their relative estimated distances to the goal. The search succeeds if a plan is found to produce the state in which the goal state is satisfied. Otherwise, the search procedure fails if the given bound is reached or the memory resources are exceeded.

The heuristic estimate is produced similarly to the aggregate algorithm as in GRT. The planner identifies all literals in the resulting state and compares them with facts in the heuristic table. Since some literals may have been used to produce other literals, the facts are grouped together by taking goal interaction into account. The grouping is performed so that the respective distances of literals are not repeatedly considered. In other words, a literal, $l$, may have been required in order to produce subsequent literals, $L$, hence the distance of $l$ to the goal should not be considered since its distance is reflected in the distances of the subsequent literals. Each group's estimate is added and the sum is returned as the heuristic value. However, to assist in guiding the search further within RBHP, this heuristic value is adjusted by the number of goal literals made true or false by the most recent action applied. Specifically, the heuristic estimate computed by referencing the table is added to the number of goal literals made true and subtracted by the number of goal literals made false by the most recent action applied.

## 4.2 RBHP Implementation

This section explains the RBHP algorithm in further detail. Specifically, the discussion expands on the techniques used to produce a solution for a planning task efficiently. Furthermore, the discussion includes the methods used to manage the procedure for particular circumstances.

**0) Preliminary Phase:**

**0.1)** Construct the binary tree representation of the initial state:

The initial state, $I$, is given as a collection of literals. Each literal is inserted into the binary tree in alphabetical order according to the object in the first argument within the literal. However, certain domains include literals which do not contain any arguments. Given such a literal, (i.e., *handempty* as in the *blocksworld* domain), then we insert an artificial object, *oracle*, into the tree. The *oracle* node is responsible for maintaining only these literals which do not possess arguments.

**0.2)** The goal state is given and maintained as a list of literals throughout the planning procedure.

**1) Regression Phase:**

**1.1)** Construct the regression graph:

Given the list of goal literals which compose the goal state, G, if all goals are satisfied, then the regression graph is initialized and returned. Otherwise, regression is applied on each goal literal, $g \in G$. Regression is applied as follows:

**i)** Given $g$, that is unsatisfied in $I$, we identify all actions, $A$, that produce $g$ as a positive effect (i.e., $\forall\ a \in A, g \in Add(a)$ ).

**ii)** In order to regress through the actions, each action's arguments are instantiated. First, the arguments within the preconditions of each action are unified with the arguments within the literal achieved by the action. The arguments within $g$ are used to instantiate the arguments

within the action's precondition because they are relevant and required by the action to produce the goal. Then the preconditions that remain with variable arguments are matched with literals that exist in $I$. Lastly, arguments within the preconditions that are unsatisfied in $I$ are instantiated by using all variations of objects given by the problem instance. In order to avoid generating irrelevant actions, each action's set of preconditions are assessed to filter actions that have conflicting preconditions. Similarly, perverse actions are filtered to remove actions that have preconditions with incorrect types for arguments as given by the action schema.

**iii)** Given multiple actions with instantiated preconditions that may produce $g$, regression is applied through each action separately. Regression is performed in order to identify the sequence of actions that produce $g$ greedily with the least amount of effort (i.e., minimum number of actions required to satisfy $g$ from $I$). It is computationally advantageous to first regress through the action which can potentially produce $g$ in a lesser number of regression steps than the subsequent actions. Once an action is shown to be rooted in $I$ by a number of regression steps, $n$, then regression through subsequent actions is halted if the number of regression steps applied exceeds $n$. Hence, prior to applying regression, we sort the actions according to the perceived effort required to establish that they are rooted in the initial state. For each $a \in A$, the perceived effort is the sum of the number of preconditions of $a$ that are not satisfied in $I$. Regression is applied on the preconditions of each action until the preconditions are found to be satisfied in the initial state. Note, all intermediate goal literals that were previously involved in regression for a particular path are maintained in order to identify cycles. Hence, the regression procedure is halted as soon as a literal that is to be regressed was involved previously since this case is identified as a cycle.

**iv)** When applying regression through an action, the action may have previously been involved in regression for another goal. Applying regression through an action previously may have been successful or unsuccessful. If successful, then the sub-graph maintained from the previous branch in the regression graph is applied to the current graph, only if, the accumulated number of steps does not exceed the bound. If unsuccessful, then regression is

performed as long as the current action layer precedes the action layer of the previous attempt. Caching previously obtained information will prevent the redundant application of regression. In turn, this memoization procedure provides a significant improvement in efficiency with regards to time.

The result for each goal is a sequence of actions, such that the final action involved in regression can be applicable in the initial state. The regression graph $\mathcal{RG}$ is similar to the planning graph $\mathcal{PG}$ produced by GRAPHPLAN. Where the $\mathcal{PG}$ consists of alternating proposition and action layers from the initial state to the goal, the $\mathcal{RG}$ consists of similar layers, but in the opposite direction. Recall that the action layers of the $\mathcal{PG}$ involve all applicable actions, given their preconditions exist in the previous proposition layer, respectively. However, the action layers of the $\mathcal{RG}$ only consist of necessary actions that achieve a proposition in the preceding proposition layer. As a result, the $\mathcal{RG}$ directly produces the actions and propositions that are essential to achieving the goal. Since most irrelevant actions applied as a product of the relaxation in the $\mathcal{PG}$ are avoided, the reachability heuristic can be extracted with less computational effort. The similarity between these graphs is made more clear by comparing Figure 2.9 with Figure 4.2. Recall that Figure 2.9 illustrates the relaxed plan sub-graph extracted from the $\mathcal{PG}$ for the given *blocksWorld* problem instance illustrated in Figure 2.6. Figure 4.2 illustrates the regression graph produced by RBHP for the same given problem given the bound is 3 steps.

For instance, the the relaxed plan sub-graph is evident within the sub-graph concerning the goal, *clear(a)*, within the regression graph. There are clear differences, whereby goals in the regression graph are shown to be achieved by more actions than shown in the relaxed plan sub-graph. Since regression is applied from a partial state to a complete state, it must consider all viable actions in the worst case in order to satisfy a goal. Whereas, the relaxed plan sub-graph is produced on the basis of mutual exclusion. Nevertheless, producing the graph by applying regression on the goal state is advantageous given that the re-computation of constructing the planning graphs as in GRAPHPLAN is avoided. Algorithm 4.1 illustrates the recursive algorithm that is responsible for producing the regression graph within the regression phase:

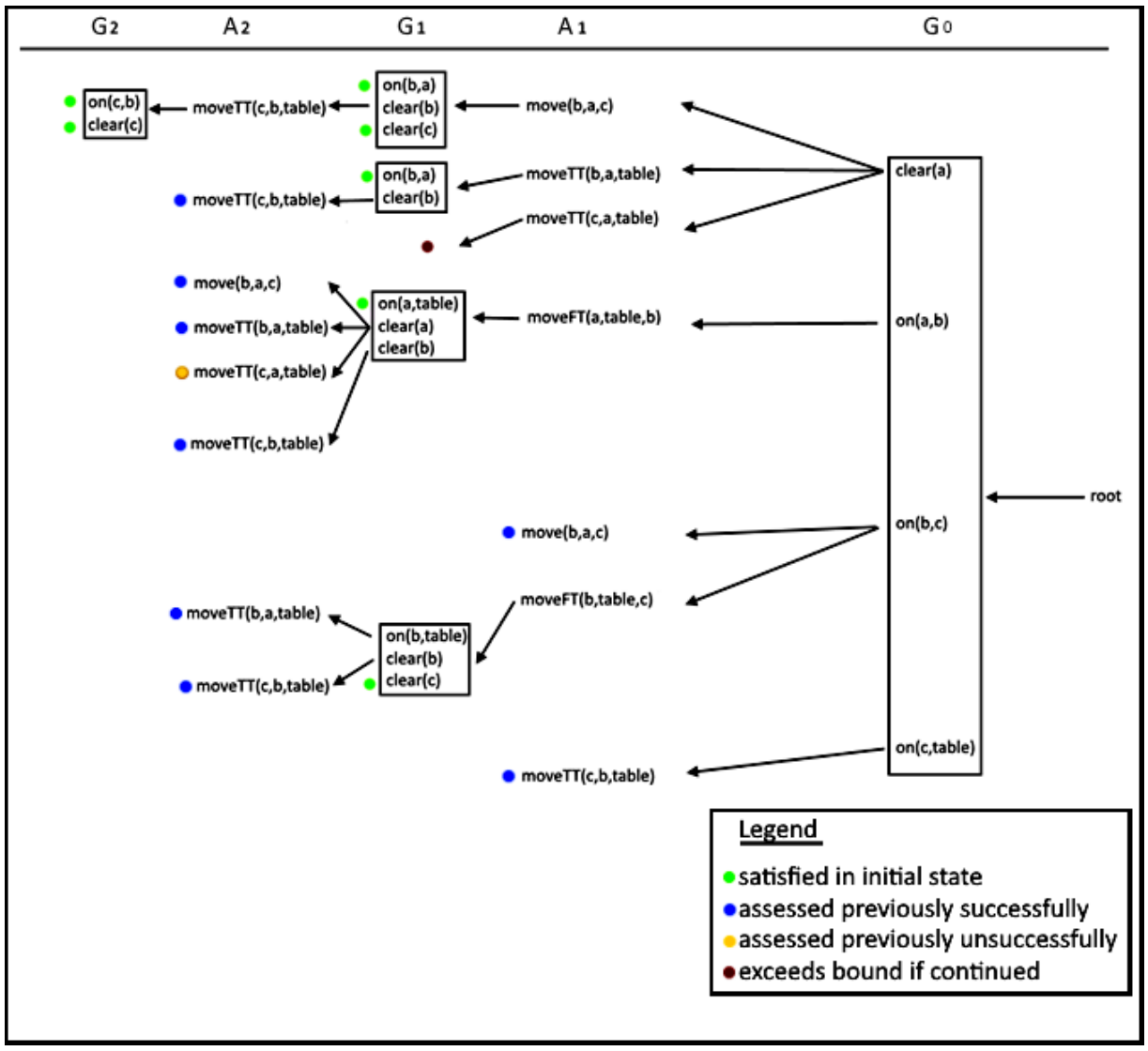



FIGURE 4.2: Illustrates the *regression graph* produced for the *blocksWorld* example in Figure 2.6

---

**Algorithm 4.1:** regGraph

---

**Input:** the initial state and goal state.

**Output:** the regression graph.

1. **regGraph:**
2. **if** all literals, $G$, are satisfied **then**
3. initialize the regression graph
4. **else**
5. **solveLiterals:** % apply multi-step regression to each literal, $g \in G$
6. **solveLiteral:**
7. identify all actions, $A$, that produce $g$
8. ground all actions $a \in A$
9. remove all perverse actions and actions with conflicting preconditions from actions $a \in A$
10. sort all grounded actions $a \in A$
11. **regress:** % apply multi-step regression through each action, $a \in A$
12. consider all preconditions of $a$ as new sub-goals, $G'$ to be regressed % as given by the action without any particular ordering
13. proceed to *step 1* unless bound limit is reached or current number of regression steps exceeds the number of steps previously able to satisfy g

---

**2) Heuristic Construction Phase:**

**2.1)** Extract the sequence of actions for each goal:

Given the regression graph, for each goal literal, $g$, of the given problem, the sequence of actions that may produce $g$ (from $I$) is extracted. Once the extraction procedure is complete, each action sequence is used to populate the heuristic table.

For instance, the following sequences of actions are extracted from the given regression graph in Figure 4.2, for the goals {*clear(a), on(a,b), on(b,c), on(c,table)*} respectively:

```
[
   [ [move(b,a,c), moveTT(b,a,table)], [moveTT(c,b,table)] ] % clear(a)
   [ [moveFT(a,table,b)], [move(b,a,c), moveTT(b,a,table)], [moveTT(c,b,table)] ] % on(a,b)
   [ [move(b,a,c)], [moveTT(c,b,table)] ] % on(b,c)
   [ [moveTT(c,b,table)] ] % on(c,table)
]
```

**2.2)** Construct Heuristic Table:

The heuristic table is motivated by the Greedy Regression Table Planning System given that it must only calculate the heuristic once. The resulting heuristic table produced by RBHP is similar to the greedy regression table produced by the GRT system. The similarity between these tables is made clear by comparing Table 2.1 with Table 4.1. Recall that Table 2.1 illustrates the greedy regression table produced by GRT system for the given *blocksWorld* problem instance illustrated in Figure 2.6. Table 4.1 illustrates the heuristic table produced by RBHP for the same given problem, however it is populated by the use of original actions as opposed to inverted actions. Subsequently, we will use the notion *table* to refer to the heuristic table used by RBHP. The resulting table stores facts that include literals which are relevant to producing the goal state. Associated with each literal is its respective distance to the goal state as well as its set of related facts. In general, the related facts of a literal, $p$, contain literals that were achieved together along with $p$, as well as the literals that were achieved previously in order to achieve $p$.

The heuristic table is initialized to contain the goal literals of the problem such that their respective distances to the goal are zero, and their respective sets of related facts are empty. Given the sequence of actions for each goal that may produce the goal literal, the negative effects of each action are inserted into the heuristic table as facts. The sequence is examined in the order such that we begin with the action that produces the goal literal directly. Subsequently, the actions further in the sequence are examined until the last action (the action applicable in the initial state) is evaluated. Associated with each fact is its distance to the goal (the number

of actions previously considered in the sequence), and its set of related facts. The role of the related facts is to promote the accurate computation of the heuristic estimate of a given state during search. Maintaining the relationship between facts prevents the redundant addition of the facts' distances to the goal state. For RBHP, the set of related facts of a literal $p$ deleted (i.e., a negative effect) by action $a$, contains:

**i) Pre(a) \Del(a)**: the literals that are preserved when applying the action (i.e., preconditions of the action which are not deleted),

**ii) rel(Add(a) ∪ Pre(a) \Del(a))**: the related facts of literals that are achieved or preserved when applying the action (i.e., the related facts of the positive effects and non-delete preconditions of the action),

**iii) Del(a)**: all other literals that are deleted by the action

**iv) \Add(a)**: without including the literals that are immediately achieved by the action.

In order to avoid the redundant addition of each fact's distance, a fact $f$ is related to previously achieved facts that were required to produce $f$ in addition to the other facts that are required together. However, facts that are immediately achieved are omitted since it is not necessary for all of them to be required to achieve the goal state. In other words, the positive effects of an action are not related facts since some effects may be irrelevant to achieving the goal. Thus, related facts only maintain literals that were necessary to produce the goal. Formally, the set of related facts to a negative effect, $p$, of an action, $a$, are as follows:

rel(p) = (Pre(a) \Del(a) ) ∪
rel( Add(a) ∪ Pre(a) \Del(a) ) ∪
Del(a) \Add(a)

Given the action sequences extracted from the regression graph, an action may be present across multiple sequences as it is required to satisfy multiple goals. In this case, a fact to be inserted into the heuristic table will be encountered repeatedly. When inserting a fact that already exists within the table, the fact with the larger distance to the goal is maintained since it must be achieved earlier during forward search. Maintaining the fact with the larger distance will in turn guide the search more accurately during the planning procedure. Algorithm 4.2 illustrates the procedure that is responsible for producing the heuristic table in further detail.

Table 4.1 illustrates the heuristic table produced given the sequences of actions extracted from the regression graph in Figure 4.2. The table consists of facts whereby each fact contains a negative effect for each action in each sequence. Each fact is associated with its respective distance to the goal and its set of related facts. Recall that in order to maintain an accurate distance to the goal state, repeated facts are resolved by preserving the fact which has the largest distance to the goal state. For instance, within Table 4.1, the fact that is furthest from the goal has a distance of 3 steps from the goal state. Note that the literal, *on(c,b)*, associated with this fact is a negative effect of the action *moveTT(c,b,table)*. As shown in the regression graph in Figure 4.2, this action is required to satisfy all goal literals that compose the goal state. In turn, this action is repeated in the sequences of actions that are extracted from the graph.

| **Facts** | **Distance to Goal** | **Related Facts** |
|---|---|---|
| clear(a) | 0 | [] |
| on(a,b) | 0 | [] |
| on(b,c) | 0 | [] |
| on(c,table) | 0 | [] |
| on(a,table) | 1 | [clear(a), clear(b)] |
| clear(b) | 1 | [clear(a), on(a,table)] |
| on(b,a) | 2 | [clear(b), on(a,table), clear(c)] |
| clear(c) | 2 | [clear(b), on(a,table), on(b,a)] |
| on(c,b) | 3 | [clear(c), on(a,table),on(b,a),clear(a)] |

TABLE 4.1: Illustrates the *heuristic table* based on the sequence of plans extracted from the *regression graph* in Figure 4.2

**Algorithm 4.2:** hsticTable

**Input:** the sequence of actions required to produce each goal within the goal state.

**Output:** the heuristic table.

1. **hsticTable:**
2. **initHsticTable:** % initialize the heuristic table with facts containing all goal literals.
3. goal literals are assigned a distance of zero to the goal state.
4. goal literals are assigned the empty set of related facts.
5. **foreach** $(A_i,...,A_n) \in A$ **do** % iterate over each sequence in the list of sequences
6. **foreach** $(a_j,...,a_n) \in A_i$ **do** % iterate over each element in the sequence
7. **if** $a_j$ is a single action **then**
8. **foreach** $(l_k,...,l_n) \in$ Del$(a_j)$ **do** % iterate over each negative effect of the action
9. *distToGoal* **is** index of $a_j$ % set distance to the goal to the index of $a_j$ within $A_i$
10. *RelFacts* **is** [] % initialize the set of related facts
11. append ( $Prec(a_j) \setminus Del(a_j)$ ) to *RelFacts*
12. append the related facts of ( $Add(a_j) \cup Prec(a_j) \setminus Del(a_j)$ ) to *RelFacts*
13. append Del$(a_j) \not\setminus \{l_i\}$ to *RelFacts*
14. remove Add$(a_j)$ from the *RelFacts*
15. **else** % aj is a list of actions
16. **foreach** $(a_m,...,a_k) \in a_j$ **do**
17. proceed to *step 8* while maintaining that *distToGoal* is the index of $a_j$
18. **return** hsticTable

Hence, to maintain the accurate distance to the goal state, the repeated fact with the largest distance to the goal is maintained since it must be achieved earlier during forward search. Specifically, since *moveTT(c,b,table)* is the third action required to satisfy the goal *on(a,b)*, its associated negative effect *on(c,b)* is attributed with a distance of 3 to the goal.

**3) Search Phase:**

**3.1)** Apply A* forward search:

Given the heuristic table, RBHP applies a situation calculus based implementation of A* search. This implementation maintains the frontier of states as nodes within a priority queue. The nodes consist of the current sequence of actions applied from the initial state which are then associated with the estimated remaining distance to the goal state. The algorithm operates by evaluating the neighbouring states in the search frontier against the heuristic table. The planner then pursues the path involving the resulting state with the lowest estimated distance to the goal state. Algorithm 4.3 illustrates the algorithm responsible for performing search.

---

**Algorithm 4.3:** plan

---

**Input:** the initial state ($I$), the goal state ($G$), and heuristic table.
**Output:** the sequence of actions that satisfies the goal from the initial state (if exists).

1. **plan:**
2. **evalState:** % estimate the distance, $d$, of the $I$ to $G$
3. $PQ$ **is** node([], $d$) % initialize the priority queue
4. **boundedAStar:**
5. identify the plan prefix, $p$, with highest priority within $PQ$
6. **if** $p$ satisfies $G$, **then**
7. **return** $p$
8. **else**
9. identify the plan prefix, $p$, with highest priority within $PQ$
10. generate the state, $S$, by applying $p$ on $I$
11. generate the neighbouring states, $nbs$, of $S$
12. **foreach** $(S'_i,...,S'_n) \in nbs$ **do**
13. estimate the heuristic value of $S'_i$ by referencing the heuristic table
14. assign the heuristic value to $p$, and insert the updated plan prefix into $PQ$
15. proceed to *step 2* unless the bound is exceeded

---

The heuristic estimate is the distance, $d$, of a resulting state to the goal state, $G$. To calculate $d$, the planner begins by extracting all literals that compose a state. Each literal is cross-referenced against the heuristic table to identify the fact that contains the literal. If a fact exists which contains the literal, it's respective distance to the goal state and related facts are determined. Otherwise, if a literal does not exist as a fact in the heuristic table, then a fact is produced to accommodate

it. The set of related facts for this fact is empty since we cannot determine the literals which were required in order to achieve this literal.

To illustrate, consider the *blocksWorld* example whereby the initial state is {*clear(c), on(c,a), on(a,table), clear(b), on(b,table)*} and the goal state is {*on(a,b)*}. Upon applying the regression phase, the resulting sequence of actions is extracted from the regression graph: [*unstack(c,a), pickup(a), stack(a,b)*]. Recall that the heuristic table contains a fact for each negative effect of each action in the sequence. Note that the literal, *on(b,table)*, is not a negative effect of any action in the given sequence. As a result, there does not exist a fact in the table that includes *on(b,table)*. However, *on(b,table)* occurs in the initial state and its respective distance to the goal state must be accounted for in order to evaluate the heuristic estimate associated with the initial state.

---

**Algorithm 4.4:** calcHsticEst

---

**Input:** list of facts of literals that compose the current state.
**Output:** the heuristic estimate.

1. **groupFacts:** % produce groups of related facts
2. *Groups* **is** []
3. **foreach** ($f_i$,...,$f_n$) $\in$ *Facts* **do**
4. *CurrentGroup* **is** [$f_i$]
5. j **is** i + 1
6. **foreach** ($f_j$,...,$f_n$) $\in$ *Facts* **do**
7. **if** $f_i \in$ relFacts($f_j$) and $f_j \in$ relFacts($f_i$) **then**
8. append(*CurrentGroup*, $f_j$) % these facts are grouped together since they have been
9. required together
10. **else if** $f_i \in$ relFacts($f_j$) and $f_j \notin$ relFacts($f_i$) **then**
11. *CurrentGroup* **is** [] % cost of $f_i$ is not included in the sum
12. **else**
13. maintain $f_i$ as a group by itself since it was not achieved by another fact $f_j$
14. **append**(*Groups*, *CurrentGroup*)
15. *hsticEst* **is** 0
16. **calcHsticEst:** % calculate the heuristic estimate
17. **foreach** ($g_i$,...,$g_n$) $\in$ *Groups* **do**
18. **add** the max cost of $g_i$ to *hsticEst*
19. **return** *hsticEst*

---

We cannot attribute this fact with an arbitrary distance to the goal since it must relate to the goal in some manner. Hence, the distance of this fact to the goal state is calculated to be the number of conflicts that the literal shares with the goal state. Specifically, it is the number of literals in the goal state which share goal objects with the literal that also contain non-goal objects. For instance, consider the goal, *on(a,b)*, and assume that *on(a,c)* is true in the current state. Then, one conflict is present given that block *a* must be on block *b* instead of block *c*. Once all the facts are produced, the heuristic estimate can be calculated.

The calculation of the heuristic estimate is illustrated in Algorithm 4.4. Briefly, facts that are related to one another are grouped together to account for goal interaction such that the distance of a literal to the goal is not redundantly added. Facts that are related to other facts (but not related to one another) are not considered in the estimation as their distance is reflected in the facts that they are related to. Otherwise, facts which are not related to any other fact are placed into distinct groups so that their distance is included in the heuristic estimate. As a result, the estimate is calculated as the sum of the maximum distance for each grouping of facts.

## 4.3 Data Set

The data set used for comparing RBHP against competing planners consists of five domains used in previous International Planning Competitions. The set was obtained in the PDDL format and translated into Prolog using our Java *PDDLToProlog* translator to conform with the binary tree representation of states. This set includes instances of problems that range in a wide range of difficulty. The problems differ with regards to the number of objects that exist as well as the number of actions which are required to solve each problem instance. Moreover, certain domains emphasize combinatorics by requiring the planner to consider several combinations of actions in order to solve the task, (i.e., *termes* and *storage* domains). Other domains involve a considerable amount of goal interaction such that goals are significantly dependent on each other to satisfy the goal state as a whole, (i.e., *blocksWorld* and *tiles*).

The following summary specifies the domains used in our data set. For each domain, a brief description of the planning task is provided, followed by a detailed explanation of the actions and fluents involved. For each action, the preconditions that must to be satisfied in order to apply

the action are provided. For each fluent, the action(s) which cause it to be true, as well as the the action(s) that cause it to be false are specified.

## Domains

### Termes

The *termes* domain involves a set of blocks and a single agent (i.e., robot) located in an area modelled as a grid. The agent is capable of creating or destroying a block only at the designated position labelled as the *depot*. The robot can manoeuvre about the area by navigating across adjacent tiles which are the same height. The robot may also navigate across adjacent tiles by climbing up or climbing down to a successive or preceding level respectively. Also, the agent can carry blocks (one at a time), and construct complex structures of towers within the space. The planning task requires the agent to construct the desired configuration of blocks within the grid-space. The *termes* domain generally involves multiple agents, however, we maintain the model of only a single agent as in the IPC-2018.

**Actions:**

- *move(PosFrom, PosTo, Height)*: the robot may move from *PosFrom* to *PosTo* iff it is at *PosFrom* that is adjacent to *PosTo*, and these positions are the same height.

- *move_up(PosFrom, HeightFrom, PosTo, HeightTo)*: the robot may move from *PosFrom* to *PosTo* iff it is at *PosFrom* that is adjacent to *PosTo* and that *HeightTo* is a successive height to *HeightFrom*.

- *move_down(PosFrom, HeightFrom, PosTo, HeightTo)*: the robot may move from *PosFrom* to *PosTo* iff it is at *PosFrom* that is adjacent to *PosTo* and that *HeightFrom* is a successive height to *HeightTo*.

- *place_block(RobotPos, BlockPos, HeightBefore, HeightAfter)*: the robot may place a block on to the position *BlockPos* iff it has a block, is at the position *RobotPos* that is adjacent to *BlockPos* and both positions are the same height.

- *remove_block(RobotPos, BlockPos, HeightBefore, HeightAfter)*: the robot may remove a block from the position *BlockPos* iff it is not holding a block, is at the position *RobotPos* that is adjacent to *BlockPos* and both positions are the same height.
- *create_block(Depot)*: the robot may create a block iff it is not holding a block, and is at the designated depot position *Depot*.
- *destroy_block(Depot)*: the robot may destroy a block iff it is holding a block, and is at the designated depot position *Depot*.

**Fluents:**

- *Height(Position, Height, s)*: the height of a *Position* becomes *Height* iff the agent places a block at *Position* or removes a block from *Position*. The height of the *Position* increases or decreases by one unit respectively. Otherwise, the *Height* of a position remains unchanged unless a block is placed at *Position* or removed from *Position*.
- *At(Position, s)*: the agent is at a *Position* iff it moves to, climbs up to, or climbs down to the *Position*. Otherwise, the agent's position remains unchanged unless it moves from, climbs up from, or climbs down from *Position*.
- *Has_block(oracle, s)*: the agent has a block iff it creates a block or removes a block from a position in the grid-space. Otherwise, the agent remains to have a block unless it destroys a block or places a block on a position in the grid-space.

**Storage**

The *storage* domain involves a set of hoists and crates located in an area represented by a grid. The hoists are capable of navigating to adjacent positions through the space which are not occupied by other hoists. Also, the hoists can carry crates (one at a time), and move them across the grid-space. The planning task is to use the given hoists to place the crates into their designated areas (i.e., containers or depots).

**Actions:**

- *lift(Hoist, Crate, CratePos, HoistPos, Place)*: the *Hoist* may lift a *Crate* from the position *CratePos* iff it is available (i.e., not lifting a crate) and is at the position *HoistPos* which is connected to *CratePos*.
- *drop(Hoist, Crate, Area, HoistPos)*: the *Hoist* may drop a *Crate* to the position *Area* iff it is lifting a *Crate*, is at the position *HoistPos*, and is connected to the *Area* which is clear.
- *move(Hoist, PosFrom, PosTo)*: the *Hoist* may move to the position *PosTo* iff it is at the position *PosFrom* that is connected to *PosTo*, and both positions are located within the same store area (i.e., either a container or a depot).
- *goOut(Hoist, PosFrom, PosTo)*: the *Hoist* may enter the loading area *PosTo* iff it is at the position *PosFrom*, that is connected to *PosTo*, whereby *PosFrom* is either a container or depot, and *PosTo* is the loading area.
- *goIn(Hoist, PosFrom, PosTo)*: the *Hoist* may enter a store area *PosTo* iff it is at the position *PosFrom*, that is connected to *PosTo*, whereby *PosFrom* is the loading area, and *PosTo* is either a container or depot.

**Fluents:**

- *Clear(Area, s)*: an *Area* is made clear iff a crate that is located in the *Area* is lifted by a hoist or a hoist that is located in *Area* moves to another space. Otherwise, an *Area* remains clear unless a crate is dropped to the *Area* by a hoist or a hoist moves to the *Area*.
- *In(Crate, Area, s)*: a *Crate* is in an *Area* iff it is dropped to the *Area* by a hoist. Otherwise, a *Crate* remains in an *Area* unless it is lifted from the *Area* by a hoist.
- *Available(Hoist, s)*: a *Hoist* is made available iff it drops a crate that it was lifting. Otherwise, a *Hoist* remains available unless it lifts a crate.
- *Lifting(Hoist, Crate, s)*: a *Hoist* is lifting a *Crate* iff it lifts the crate. Otherwise, a *Hoist* remains to be lifting a *Crate* unless it drops the *Crate* it was lifting.

- *At(Hoist, Position, s)*: a *Hoist* is at a *Position* iff it moves to the *Position*. Otherwise, a *Hoist* remains to be at a *Position* unless it moves from the *Position*.

- *On(Crate, Container, s)*: A *Crate* is on a *Container* iff it is dropped onto the *Container*. Otherwise, a *Crate* remains to be on a *Container* unless it is lifted from the *Container* by a hoist.

**BlocksWorld**

The *blocksWorld* domain is one of the most well-known domains in classical planning. This domain consists of a set of blocks, a single table, and a single robotic arm. A block can only be on top of another block or directly on the table. Given an initial configuration of blocks, the arm is used to manipulate the blocks (one at a time) in order to produce the desired configuration.

**Actions:**

- *pickup(BlockA)*: *BlockA* may be picked up iff the arm is free (i.e., not holding a block), *BlockA* is clear, and *BlockA* is on the table.

- *putdown(BlockA)*: *BlockA* may be put down on to the table iff the arm is holding *BlockA*.

- *stack(BlockA,BlockB)*: *BlockA* may be stacked on top of *BlockB* iff the arm is holding *BlockA*, and *BlockB* is clear.

- *unstack(BlockA,BlockB)*: *BlockA* may be unstacked from *BlockB* iff *BlockA* is on top of *BlockB*, *BlockA* is clear, and the arm is free.

**Fluents:**

- *On(BlockA, BlockB, s)*: *BlockA* is on top of *BlockB* iff *BlockA* is stacked on *BlockB*. Otherwise, *BlockA* remains on top of *BlockB* unless it was unstacked from *BlockB*.

- *Ontable(BlockA, s)*: *BlockA* is directly on the table iff *BlockA* is put down on to the table. Otherwise, *BlockA* remains on the table unless it was picked up.

- *HandEmpty(oracle, s)*: the arm is free iff it put down a block on to the table or stacked a block on to another block. Otherwise, the arm remains to be free unless a block is picked up from the table or a block is unstacked from another block.

- *Holding(BlockA, s)*: the arm is holding *BlockA* iff it picked up a block from the table or unstacked a block from another block. Otherwise, the arm remains to hold *BlockA* unless *BlockA* is put down on to the table or stacked onto another block.

**Tiles**

The *tiles* domain is also one of the most well-known domains in classical planning. This domain involves a grid of numbered tiles, whereby one tile is missing (i.e., labelled as the empty *space* tile). The planning task is to arrange the tiles by (repeatedly) swapping a tile adjacent to the *space* tile in order to produce the desired configuration.

**Actions:**

- *move(Tile, PosFrom, PosTo)*: a *Tile* may be moved iff it is adjacent to the *space* tile.

**Fluents:**

- *Pos(Position, Tile, s)*: a *Tile* is at a *Position* iff it is moved to the *Position*. Otherwise, a *Tile* remains to be at a *Position* unless it is moved away from the *Position*.

**Gripper**

The *gripper* domain involves a set of balls, and an agent (i.e., robot) located in an area modelled as a grid of rooms. The agent has two arms available to pick up or drop a ball. The agent can carry balls (one ball in each arm at most) and can navigate to and from adjacent rooms within the space.

**Actions:**

- *move(PosFrom, PosTo)*: the robot may move from *PosFrom* to *PosTo* iff it is at the room *PosFrom* that is connected to the room *PosTo*.
- *pick(Ball, Room, Robot)*: the *Robot* may pick up a *Ball* iff it has at least one free arm (i.e., not holding a ball), and both the *Ball* and *Robot* are in the same *Room*.
- *drop(Ball, Room, Robot)*: the *Robot* may drop a *Ball* in a *Room* iff it is carrying a *Ball*.

**Fluents:**

- *AtRobot(Room, s)*: the robot is at a *Room* iff it moves to the *Room*. Otherwise, the robot remains to be at a *Room* unless it moves away from the *Room*.
- *AtObject(Ball, Room, s)*: a *Ball* is at a *Room* iff it is dropped at the *Room*. Otherwise, a *Ball* remains to be at a *Room* unless it is picked up by the robot from the *Room*.
- *Free(RobotArm, s)*: a *RobotArm* is free iff it drops a ball that it was carrying. Otherwise, a *RobotArm* remains to be free unless it picks up a ball.
- *Carrying(Ball, Robot, s)*: the *Robot* is carrying a *Ball* iff it picks up a *Ball*. Otherwise, the *Robot* remains to be carrying a *Ball* unless it drops the *Ball*.

Note that the set of problems chosen is restricted to only include domains which do not involve conditional effects as part of their action schemas. Domains involving conditional effects for actions increases the complexity of regression since the conditions must be included in the regression procedure. Applying regression on the literals included in the conditions of effects will increase the branching factor of the regression graph. Hence, the planner was not tested on the subset of domains which involve conditional effects until significant results could be produced on domains that do not involve conditional effects. Additionally, the domains consist only of relational fluents.

## 4.4 Experimental Results

The following tables evaluate RBHP on several domains involved in previous planning competitions. For our purposes, results of planners are deemed to be competitive according to the standard metrics used in the planning competitions. Thus, given that the planner lacks competitive performance against competing planners, the results are displayed only for RBHP. The results are produced under the following specifications: i5, single CPU core, @2.50GHz, 2GB RAM whereby RBHP was given a time limit of 1 hour to produce a solution. Note that the column displaying the performance of RBHP shows the time in seconds required for the planner to produce a solution to the given problem instance. Otherwise, "na" signifies that a solution cannot be produced within the time limit.

| Domain | Problem ID | Number of Objects | Optimal Number of Steps Required | RBHP (sec) |
|---|---|---|---|---|
| **Storage** | 1 | 7 | 3 | 0 |
| | 2 | 9 | 3 | 0 |
| | 3 | 11 | 3 | 0.11 |
| | 4 | 12 | 8 | 0.07 |
| | 5 | 13 | 8 | 1.17 |
| | 6 | 14 | 8 | 1.31 |
| | 7 | 16 | 14 | 4.41 |
| | 8 | 17 | 14 | na |
| | 9 | 18 | 12 | na |
| | 10 | 20 | 18 | na |

TABLE 4.2: Experimental results produced by RBHP on the *storage* domain

| Domain | Problem ID | Number of Objects | Optimal Number of Steps Required | RBHP (sec) |
|---|---|---|---|---|
| **Tiles_3x3** | 1 | 9 | 8 | 0.09 |
| | 2 | 9 | 9 | 1.36 |
| | 3 | 9 | 10 | 1.31 |
| | 4 | 9 | 11 | 2.36 |
| | 5 | 9 | 12 | 4.01 |
| **Tiles_6x6** | 1 | 36 | 7 | 0.03 |
| | 2 | 36 | 8 | 0.06 |
| | 3 | 36 | 9 | 0.08 |
| | 4 | 36 | 10 | 2.31 |
| | 5 | 36 | 12 | 3.42 |
| | 6 | 36 | 16 | na |
| | 7 | 36 | 20 | na |

TABLE 4.3: Experimental results produced by RBHP on the *tiles* domain

| Domain | Problem ID | Number of Objects | Optimal Number of Steps Required | RBHP (sec) |
|---|---|---|---|---|
| **BlocksWorld** | 4-0 | 4 | 6 | 0.06 |
| | 4-1 | 4 | 10 | 0.23 |
| | 4-2 | 4 | 6 | 0.02 |
| | 5-0 | 5 | 12 | 0.11 |
| | 5-1 | 5 | 10 | 0.3 |
| | 5-2 | 5 | 16 | 1.91 |
| | 6-0 | 6 | 12 | 14.02 |
| | 6-1 | 6 | 10 | 0.09 |
| | 6-2 | 6 | 20 | na |
| | 7-0 | 7 | 20 | na |
| | 7-1 | 7 | 22 | na |
| | 7-2 | 7 | 20 | na |
| | 8-0 | 8 | 18 | na |
| | 8-1 | 8 | 20 | na |
| | 8-2 | 8 | 16 | 16.16 |
| | 9-0 | 9 | 30 | na |
| | 9-1 | 9 | 28 | na |
| | 9-2 | 9 | 26 | na |
| | 10-0 | 10 | 32 | na |
| | 17-0 | 17 | 46 | na |

TABLE 4.4: Experimental results produced by RBHP on the *blocksWorld* domain

| Domain | Problem ID | Number of Objects | Optimal Number of Steps Required | RBHP (sec) |
|---|---|---|---|---|
| **Termes** | 1 | 16 | 4 | 0.98 |
| | 2 | 16 | 8 | 7.36 |
| | 3 | 16 | 18 | na |
| | 4 | 16 | 36 | na |

TABLE 4.5: Experimental results produced by RBHP on the *termes* domain

The following table compares RBHP against the top three scoring planners submitted to the IPC-2018 on the *gripper* domain. These planners, namely *FD-Stone-Soup*, *FD-Remix*, and *LAPKT-DUAL-BFWS* among other planners submitted to the IPC-2018 have been developed for over 10 years by leading AI planning research centers who focus largely on the area of automated planning [23, 41]. These research groups consist of senior PhD students and post doctoral researchers. Whereas, RBHP has only been in development for less than one year. The results are produced under the following specifications: i5, single CPU core, @2.50GHz, 2GB RAM whereby the planners where given a time limit of 1 hour to produce a solution. Note that the column displaying the performance of each planner shows the time in seconds required for the planner to produce a solution to the given problem instance.

| **Domain** | **Problem ID** | **Number of Objects** | **Optimal Number of Steps Required** | **RBHP (sec)** | **FD-Stone-Soup (team_45) (sec)** | **FD-Remix (team_43) (sec)** | **LAPKT-DUAL_ BFWS (team_20) (sec)** |
|---|---|---|---|---|---|---|---|
| **Gripper** | 1 | 4 | 11 | 0 | 0 | 0 | 0 |
| | 2 | 6 | 17 | 0 | 0 | 0 | 0 |
| | 3 | 8 | 23 | 0.04 | 0 | 0 | 0 |
| | 4 | 10 | 29 | 0.06 | 0 | 0.01 | 0.01 |
| | 5 | 12 | 35 | 0.07 | 0 | 0.01 | 0.01 |
| | 6 | 14 | 41 | 0.12 | 0 | 0.01 | 0.01 |
| | 7 | 16 | 47 | 0.14 | 0.01 | 0.01 | 0 |
| | 8 | 18 | 53 | 0.23 | 0.01 | 0.01 | 0.01 |
| | 9 | 20 | 59 | 0.34 | 0.01 | 0.01 | 0.02 |
| | 10 | 22 | 65 | 0.37 | 0.01 | 0.01 | 0.02 |
| | 11 | 24 | 71 | 0.53 | 0.02 | 0.02 | 0.03 |
| | 12 | 26 | 77 | 0.64 | 0.02 | 0.02 | 0.04 |
| | 13 | 28 | 83 | 0.79 | 0.02 | 0.02 | 0.04 |
| | 14 | 30 | 89 | 1.03 | 0.02 | 0.02 | 0.05 |
| | 15 | 32 | 95 | 1.15 | 0.02 | 0.02 | 0.07 |
| | 16 | 34 | 101 | 1.45 | 0.03 | 0.02 | 0.08 |
| | 17 | 36 | 107 | 1.71 | 0.03 | 0.03 | 0.09 |
| | 18 | 38 | 113 | 2.01 | 0.04 | 0.03 | 0.1 |
| | 19 | 40 | 119 | 2.29 | 0.04 | 0.04 | 0.12 |
| | 20 | 42 | 125 | 2.71 | 0.04 | 0.04 | 0.14 |

TABLE 4.6: Competitive results produced by RBHP against IPC planners on the *gripper* domain

## 4.5 Analysis of Results

As shown in Section 4.4 regarding the experimental results, RBHP performs competitively for domains that possess certain characteristics. For instance, with respect to the optimality and satisficing tracks in IPCs, all planners would receive the same score for finding the optimal solutions to all problem in the *gripper* domain. With respect to the metric used in the agile track, all planners would receive the same score of 1 for solving problems 1 to 13 in less than 1 second. The state-of-the-art planners would continue to receive a score of 1 for the remaining problems, however, RBHP would receive a scaled comparable factor.

RBHP suffers from two main constraints. The first limitation is present during the regression phase. Upon identifying a set of actions that produce a goal as a positive effect, the arguments of the actions must be instantiated in order to apply regression on the actions' preconditions. However, certain domains involve a large number of actions or a large number of variations of actions that can produce the desired goal. Subsequently, regression must be applied through these actions in order to identify the path which greedily satisfies the goal. As in Unpop and GRT, these algorithms must exhaustively apply regression or search to identify the greedy solution for each goal. Although memoization used to cache the information obtained in regression previously offers a significant improvement, RBHP exceeds the available memory for difficult problems of certain domain. In turn, the procedure terminates without finding a solution to the given problem.

For instance, within the *termes* domain, the goal, *at(Position)*, can be made true if the robot moved to the *Position* from one of the adjacent positions at the same height. Additionally, *at(Position)* can be made true if the robot climbed up from an adjacent position that is one level lower than the *Position*. Similarly, *at(Position)* can be made true if the robot climbed down from an adjacent position that is one level higher than the *Position*. Given the large number of variations of actions that can satisfy a given goal, RBHP is unable to produce a solution.

The second limitation of RBHP is experienced during the search phase. Recall that the goal literals are solved individually during the regression phase, upon which the sequence of actions that may produce each goal literal is extracted. These sequences are used in populating the heuristic table. The actions are evaluated in the order from the action which produces the goal directly, to the action that is applicable in the initial state. During the heuristic construction phase, the

table will be populated with facts containing the negative effects of actions. In turn, the table will include a tentative solution path (i.e., literals that must be achieved) for each goal literal. However, when populating the heuristic table, the sequences are used without taking goal ordering into account. Hence, the resulting heuristic table will guide the search in such a way that goals are satisfied without precedence. In other words, during the search, each resulting state in the frontier that progresses towards satisfying a goal is equally pursued. As a result, RBHP lacks competitive performance for planning tasks which require certain goals to be achieved before others (i.e., strict goal order).

For instance, the heuristic table may lead the planner to build the upper tiers of a block tower from *blocksWorld* before building the foundation. In turn, building the tower out of order will cause the planner to pursue an incorrect path early in the search. Since an incorrect action is chosen early in the planning phase, more time is spent in identifying a solution. The planner continues to generate and maintain additional nodes within the search frontier. In turn, this excessive search leads the planner to continue searching until the time limit is reached or it exceeds the available memory. Conversely, the planner does not suffer from this limitation on planning tasks with goals that are not as dependent on each other, since they do not require strict goal ordering.

As a result, RBHP performs well on domains such as the *gripper* domain since the order of satisfying the goals is irrelevant. Recall the planning task of the *gripper* domain requires a robot to move balls from their present location to the desired location. Specifically, moving a particular ball from one room to another room does not depend on the other balls. As a result, RBHP performs competitively against planners submitted to the IPC-2018 for certain domains.

As an aside, for competing planners that perform grounding, we attempt to show that increasing the number of objects within the state will prevent the planner from performing search. For instance, we add several thousand objects to instances from the *gripper* domain that are irrelevant to the search and demonstrate that RBHP continues to produce significant results. Nevertheless, as shown within the raw output, these planners that perform grounding are able to identify and remove auxiliary atoms when producing the grounded transition system. As a result these planners are unaffected by the increase in the number of irrelevant objects. However, as the number of irrelevant objects are increased, RBHP will exhibit poorer performance. This is due to the two main reasons. First, regression will be applied through actions that may involve these irrelevant

objects. In turn, more time is required to regress through the increased number of actions that may make a goal true. The second reason is due to the additional time required to access and update fluents within the binary tree representation of states. Given that the number of objects are increased, the tree must maintain nodes for these objects and cause RBHP to take more time in general to assess preconditions of actions or modify the state as a result of an action being applied.

# Chapter 5

# Conclusion and Future Work

## 5.1 Summary and Contribution

In this thesis, we explore the limitations of the grounding procedure exhibited by modern state-of-the-art automated planners. Specifically, we produce the Organic Chemistry Synthesis benchmark based on Organic Chemistry examination questions from the Massachusetts Institute of Technology. This benchmark was encoded in PDDL and proposed as a new challenging benchmark to the latest International Planning Competition (2018). This set of problems was used in gauging the performance of competing planners, demonstrating that grounding is an issue. This benchmark was awarded with the *Outstanding Domain Submission Award* by the organizers of the competition for our effort.

In order to integrate heuristics and maintain efficiency when solving planning tasks, we develop a new BinTree solver. This solver provides an approach for representing the states and action schemas which takes advantage of the binary tree data structure. The BinTree solver, in junction with an efficient implementation of sub-graph isomorphism, was shown to produce a solution for 7 out of the 20 organic chemistry synthesis problems. This performance is comparable with the state-of-the art planners.

Additionally, we developed a novel domain-independent planner, namely *RBHP*. RBHP is designed to solve planning tasks without the need to instantiate the entire the set of actions before conducting search. To operate efficiently, RBHP performs (goal-directed) backward heuristic construction based on goal-regression. Subsequently, the actions which were identified to satisfy the goals are then used to populate the heuristic table to provide heuristics during forward A* search.

This planner builds on various advancements in planning made by the AI community. As in GRAPHPLAN, RBHP produces the reachability heuristic. However, RBHP efficiently achieves this by performing depth-first goal regression on the goal state, as in Unpop. Hence, the planner computes the reachability heuristic without producing the forward planning graphs. As in GRT, our planner takes advantage of the greedy regression table to provide heuristics during forward search [61]. However, RBHP populates the table by using the given actions through regression as opposed to using inverted actions. Unlike GRT, this feature enables RBHP to efficiently solve problem instances that involve incomplete goal states. Furthermore, RBHP uses a priority queue to maintain plan prefixes throughout the planning phase. Maintaining plan prefixes instead of states within the search frontier reduces the burden on memory resources.

We also presented and analysed the competitive results produced by RBHP against competing planners. The results demonstrate potential, in that, our planner is able to compete with state-of-the-art planners without the need to build a grounded transition system prior to searching for a solution. Although RBHP is able to produce some significant results, it experiences several limitations. Recall that RBHP suffers from identifying the sequence of actions which can satisfy a given goal greedily for certain domains. These domains involve a significant number of variations of actions that may lead to satisfying the goal. In turn, the branching factor may prevent RBHP from completing the regression phase despite the effort applied to alleviate the inefficiency (i.e., memoization).

Furthermore, recall that the heuristic is constructed by first producing a solution for each individual goal literal included in the goal state. Goal interaction and goal ordering are not taken into account throughout the regression phase. As a result, the resulting heuristic table suffers from not identifying the necessary ordering required to solve the goal state as a whole. Therefore, at this time, RBHP can consistently guide the search accurately only for certain domains that do not involve strict goal ordering.

## 5.2 Future Work

While RBHP demonstrates potential, it suffers from several limitations as demonstrated by the analysis of the experimental results. During the regression phase, the planner could exploit a method of sorting actions more accurately prior to applying regression. As a result, the solution that achieves a given goal greedily will be produced earlier. In turn, this will speed up the procedure for the regression through subsequent actions. Additionally, further research should be spent in applying regression through abstract actions without instantiated arguments. In turn, regressing through these high-level actions with variable arguments has potential in alleviating the constraint created by applying regression through all variations of grounded actions. However, this approach gives rise to another bottleneck. The high-level actions must be ground at some point to produce a solution for the goals. If the initial state consists of a large number of objects, then grounding the actions exhaustively from the initial state will require a significant amount of time.

Additionally, the trade-off between constructing the heuristic only once and computing the heuristic at each stage of forward search should be explored further. Planners such as Unpop and GRAPHPLAN compute the heuristic at each step in forward search given that planning is performed from a complete state towards a partial goal state. For instance, the planning graphs produced by GRAPHPLAN must be recomputed as there may exist multiple goal states that consist of the goal propositions in addition to side effects produced by actions in the plan. Hence, the planning graphs must be computed at each step in forward search to produce the heuristic respective to the current state. Also, the regression-match and planning graphs are re-computed during search given that they are not maintained, unlike in the GRT system which maintains the greedy regression table after constructing it only once. The heuristic estimate produced by Unpop and GRAPHPLAN are likely to be more accurate with regards to the current state as opposed to the GRT and RBHP which compute the heuristic with regards to the initial state only once. However, at the cost of producing a less informative heuristic, the GRT and RBHP planners preserve time and computational effort by avoiding the redundant construction of the heuristic. Mats Pettersson [55] discusses this trade-off in further detail by comparing planners which construct the planning graphs using regression and progression across several domains.

Furthermore, during the heuristic construction phase, RBHP would benefit by ordering the individual plans (for each goal) prior to populating the heuristic table. Given that the actions are ordered correctly, then the resulting heuristics table will be better able to guide the search. Additionally, given the complexity of RBHP, the main focus was aimed at solving the subset of domains that do not involve conditional effects. Future work involves expanding applicability of RBHP to the domains with context-dependent effects. Furthermore, we have produced implementations of several planners that offered insight or potential advantages with regards to regression planning: *R*, *MM*, *RO*, *Unpop*, and *GRT*. These planners can be revisited to identify beneficial techniques.

In regards to the Organic Synthesis benchmark, additional improvements can be made. In order to compensate for the difficulty of identifying arguments for the preconditions of actions, the sub-graph isomorphism algorithm was implemented. Recall that this algorithm operates by identifying functional groups as opposed to satisfying the conjunction of literals in the preconditions of actions. Since, the preconditions cannot be efficiently grounded using regression, RBHP cannot be used to solve the problems related to the organic synthesis benchmark. If an efficient domain-independent method can be developed in order to ground the preconditions of actions, then RBHP can be applied to the organic synthesis domain as well. Additionally, while the Organic Synthesis benchmark is chemically accurate in two-dimensional space, it can be improved to better represent the chemical compounds and bonds in accordance with reality. Through collaborations with experts in chemistry, as well as the use of verified online resources, the bonds between atoms and molecules can be further elaborated by representing them in three-dimensional space. Syntactically, minor adjustments must be made to the bond predicates within the initial and goal states. Specifically, an additional argument which dictates the orientation of the bond must be introduced. Additionally, the encoding of functional groups must also be adjusted in a similar manner. Furthermore, the sub-graph isomorphism implementation must be amended by performing an additional assessment (i.e., within the syntactic feasibility rules). This assessment will enforce the matching of corresponding configurations of bonds in three-dimensional space. Intuitively, this change in representation will improve the algorithm's efficiency. Given that the molecules are represented in three-dimensional space, the preconditions of actions can be identified more efficiently as sub-graph isomorphism will more efficiently match the appropriate set of functional groups.

# References


[1] Christer Bäckström and Peter Jonsson. All PSPACE-complete planning problems are equal but some are more equal than others. In Daniel Borrajo, Maxim Likhachev, and Carlos Linares López, editors, *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*, pages 10–17. AAAI Press, 2011.

[2] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1636–1642. Morgan Kaufmann, 1995.

[3] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

[4] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

[5] Daniel Bryce and Subbarao Kambhampati. A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(1):47–83, 2007.

[6] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

[7] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the VF graph matching algorithm. In *10th International Conference on Image Analysis and Processing (ICIAP 1999), 27-29 September 1999, Venice, Italy*, pages 1172–1177. IEEE Computer Society, 1999.

[8] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[9] Elias J. Corey. *The logic of chemical synthesis*. Nobel Foundation [Nobelstiftelsen], 1991.

[10] Elias J. Corey and Todd Wipke. Computer-assisted design of complex organic syntheses. *Science*, 166(3902):178–192, oct 1969.

[11] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, July 1985.

[12] Fred Dushin. Java-to-prolog (jpl) interface, 2003. Available at: `https://www.swi-prolog.org/packages/jpl/java_api/index.html`.

[13] Christian Ehrlich and Matthias Rarey. Systematic benchmark of substructure search in molecular graphs - from Ullmann to VF2. *Journal of cheminformatics*, 4:13, 07 2012.

[14] George Ernst and Allen Newell. *GPS: a case study in generality and problem solving*. ACM monograph series. Academic Press, 1969.

[15] Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan Sturtevant, and Zhifu Zhang. Inconsistent heuristics in theory and practice. *Artificial Intelligence*, 175(9):1570 – 1603, 2011.

[16] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[17] Maria Fox and David Poole, editors. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[18] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2013.

[19] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *CoRR*, abs/1110.2737, 2011.

[20] Peter E. Hart, Nils. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[21] Abraham Heifets. Benchmark problems, 2012. Available at: `http://www.cs.toronto.edu/~aheifets/ChemicalPlanning/`.

[22] Abraham Heifets. Construction of new medicines via game proof search. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012.

[23] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[24] Jörg Hoffmann. FF: the fast-forward planning system. *AI Magazine*, 22(3):57–62, 2001.

[25] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001.

[26] Robert C. Holte, Ariel Felner, Guni Sharon, Nathan R. Sturtevant, and Jingwei Chen. MM: A bidirectional search algorithm that is guaranteed to meet in the middle. *Artif. Intell.*, 252:232–266, 2017.

[27] Jochem Hüllen and Frank Weberskirch. Extracting goal orderings to improve partial-order planning. In *Proceedings PuK 99, 13. Workshop "Planen und Konfigurieren", Jahrestreffen der Fachgruppe 1.5.3 der Gesellschaft für Informatik, vom 3.-5. März 1999, im Rahmen der 5. Tagung "Expertensysteme" an der Universität Würzburg*, 1999.

[28] Subbarao Kambhampati, editor. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. IJCAI/AAAI Press, 2016.

[29] Subbarao Kambhampati and Romeo Sanchez Nigenda. Distance-based goal-ordering heuristics for graphplan. In Steve A. Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pages 315–322. AAAI, 2000.

[30] Michael Katz and Carmel Domshlak. Optimal admissible composition of abstraction heuristics. *Artif. Intell.*, 174(12-13):767–798, 2010.

[31] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *J. Artif. Intell. Res.*, 12:338–386, 2000.

[32] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[33] Richard E. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990.

[34] Richard E. Korf. Linear-space best-first search. *Artif. Intell.*, 62(1):41–78, July 1993.

[35] Richard E. Korf. Artificial intelligence search algorithms. In *In Algorithms and Theory of Computation Handbook*. CRC Press, 1996.

[36] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2016.

[37] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[38] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[39] Fangzhen Lin. An ordering on subgoals for planning. *Ann. Math. Artif. Intell.*, 21(2-4):321–342, 1997.

[40] Fangzhen Lin. A planner called R. *AI Magazine*, 22(3):73–76, 2001.

[41] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3590–3596. AAAI Press, 2017.

[42] Ambuj Mahanti, Subrata Ghosh, Dana Nau, Asim Pal, and Laveen Kanal. On the asymptotic performance of IDA*. *Annals of Mathematics and Artificial Intelligence*, 20:161–193, 01 1997.

[43] Zohar Manna and Richard J. Waldinger. Knowledge and reasoning in program synthesis. *Artif. Intell.*, 6(2):175–208, 1975.

[44] Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling organic chemistry and planning organic synthesis. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 176–195. EasyChair, 2015.

[45] Arman Masoumi and Mikhail Soutchanski. Reasoning about chemical reactions using the situation calculus. In *Discovery Informatics: The Role of AI Research in Innovating Scientific Processes, Papers from the 2012 AAAI Fall Symposium, Arlington, Virginia, USA, November 2-4, 2012*, volume FS-12-03 of *AAAI Technical Report*. AAAI, 2012.

[46] Arman Masoumi, Mikhail Soutchanski, and Andrea Marrella. Organic synthesis as artificial intelligence planning. In Adrian Paschke, Albert Burger, Paolo Romano, M. Scott Marshall, and Andrea Splendiani, editors, *Proceedings of the 6th International Workshop on Semantic Web Applications and Tools for Life Sciences, Edinburgh, UK, December 10, 2013.*, volume 1114 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[47] Rami Matloob and Mikhail Soutchanski. Exploring organic synthesis with state-ofthe-art planning techniques. 2016.

[48] John McCarthy. Situations, actions, and causal laws. Technical report, Stanford University Computer Science, 1963.

[49] Ciaran McCreesh, Patrick Prosser, and James Trimble. Heuristics and really hard instances for subgraph isomorphism problems. In Kambhampati [28], pages 631–638.

[50] Drew V. McDermott. A heuristic estimator for means-ends analysis in planning. In Brian Drabble, editor, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, May 29-31, 1996*, pages 142–149. AAAI, 1996.

[51] Drew V. McDermott. Using regression-match graphs to control search in planning. *Artif. Intell.*, 109(1-2):111–159, 1999.

[52] Drew V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

[53] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[54] Edwin P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15-18 1989.*, pages 324–332. Morgan Kaufmann, 1989.

[55] Mats Petter Pettersson. Reversed planning graphs for relevance heuristics in AI planning. 2005.

[56] John L. Pollock. The logical foundations of goal-regression planning in autonomous agents. *Artif. Intell.*, 106(2):267–334, 1998.

[57] Florian Pommerening, Malte Helmert, Gabriele Röger, and Jendrik Seipp. From non-negative to general operator cost partitioning. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3335–3341. AAAI Press, 2015.

[58] Orr Ravitz. Data-driven computer aided synthesis design. *Drug Discovery Today: Technologies*, 10:e443–e449, 09 2013.

[59] Igor Razgon and Ronen I. Brafman. A forward search planning algorithm with a goal ordering heuristic. In *Proceedings of the 6th European Conference on Planning, September 12-14, Toledo, Spain*, pages 25 – 36, 2001.

[60] Ioannis Refanidis and Ioannis P. Vlahavas. GRT: A domain independent heuristic for STRIPS worlds based on greedy regression tables. In Susanne Biundo and Maria Fox, editors, *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999, Proceedings*, volume 1809 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 1999.

[61] Ioannis Refanidis and Ioannis P. Vlahavas. The GRT planning system: Backward heuristic construction in forward state-space planning. *J. Artif. Intell. Res.*, 15:115–161, 2001.

[62] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[63] Silvia Richter, Jordan Tyler Thayer, and Wheeler Ruml. The joy of forgetting: Faster anytime search via restarting. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, pages 137–144. AAAI, 2010.

[64] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[65] Punnaivanam Sankar, Alain Krief, and Durairaj Vijayasarathi. A conceptual basis to encode and detect organic functional groups in XML. *Journal of molecular graphics and modelling*, 43C:1–10, 04 2013.

[66] Roni Stern, Tamar Kulberis, Ariel Felner, and Robert Holte. Using lookaheads with optimal best-first search. In Fox and Poole [17].

[67] Sara Szymkuć, Ewa P. Gajewska, Tomasz Klucznik, Karol Molga, Piotr Dittwald, Michał Startek, Michał Bajczyk, and Bartosz A. Grzybowski. Computer-assisted synthetic planning: The end of the beginning. *Angewandte Chemie International Edition*, 55(20):5904–5937, 2016.

[68] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[69] Dimitris Vrakas, Dimitris Vrakas, and Ioannis Pl Vlahavas. *Artificial Intelligence for Advanced Problem Solving Techniques*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2008.

[70] Richard Waldinger. Achieving several goals simultaneously. Technical Report 107, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Jul 1975.

[71] Rong Zhou and Eric A. Hansen. Memory-bounded a* graph search. In Susan M. Haller and Gene Simmons, editors, *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, May 14-16, 2002, Pensacola Beach, Florida, USA*, pages 203–209. AAAI Press, 2002.